RealSpeak Telecom
Software Development Kit

SpeechWorks® solutions
from ScanSoft®

RealSpeak v4.0 Manual

Notice

Copyright © 1995-2005 by ScanSoft, Inc. All rights reserved.
ScanSoft, Inc. provides this document without representation or warranty of any kind. ScanSoft, Inc. reserves the right to revise this document and to change the information contained in this document without further notice.
Realspeak, DialogModules, OpenSpeech, Productivity Without Boundaries, ScanSoft, the ScanSoft logo, SMARTRecognizer,SpeechCare, Speechify, SpeechSecure, SpeechSpot, SpeechSite, SpeechWorks, the SpeechWorks logo, and SpeechWork-sHere are trademarks or registered trademarks of ScanSoft, Inc. or its licensors in the United States and/or other coun-tries.
Portions of the OpenSpeech Recognizer Software are subject to copyrights of AT&T Corp., E-Speech Corporations, Bell Communications Research, Inc., European Telecommunications Standards Institute and GlobeTrotter Software, Inc. GoAhead WebServer Copyright © 2004 GoAhead Software, Inc. All Rights Reserved.
U.S. Patent Nos. 5,634,087; 5,839,103; 5,862,519; 5,995,928; 5,809,494; 5,765,130; 6,061,651; 6,173,266; 6,519,561; 6,539,352; US6665641 and EP1501075. One or more patents may be pending in the United States and other countries.
Without limiting the rights under copyright reserved above, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means, including, without limitation, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of ScanSoft, Inc.

Published by:
ScanSoft, Inc.
Worldwide Headquarters
9 Centennial Drive
Peabody, MA 01960
United States

RealSpeak Telecom for Windows Software Development Kit User's Guide and Programmer's Reference
V4.0 © - December 2005

Table of Contents

INTRODUCTION 16

Introduction to RealSpeak 16

Organization of this manual 16

Contacting ScanSoft 17
Defect Report Form 17

System Overview 17
Introduction 17
API Support 18
Markup Support 18
Product Support 18
Input/Output behavior of RealSpeak 19
Three different text input techniques 19
Presentation of the input text 19
Language and voice switching 19
Audio output streaming 19
Modes of operation: in-process and Client/Server 20
In-process mode 20
Client-Server mode 22

Use of RealSpeak in telephony environments 27

New features for RealSpeak 4.0 28

INSTALLATION GUIDE 31

Licensing 31
Licensing - Important note 31
Overview of licensing 31

Installation on Windows 32
Installation Steps for Windows 32
Install the common installer 32
Install the voice specific installer 35
Configuring the licensing 37
Running a demo program 38

Installation on Linux 39
Installation Steps for Linux 39
Step 1: Install the common components 39
Step 2: Install the purchased voices 39
Step 3: Install the licensing components 39
Step 4: Configuring and starting of the licensing 39
Step 5: Updating your environment settings 40
Step 6: Running a sample program 41

Environment variables 42

RealSpeak Components 43

RealSpeak API library 43
TTS API support libraries 43
TTS server 44
Engine and language libraries 44
Demo programs 44

DEPLOYING REALSPEAK 46

Introduction 46

In-process use of RealSpeak 47
Intro 47
API Call Sequence 47
Demonstration applications 49
Standard Demo 49
Some comments on the implementation 50
Standardex Demo 50
Some comments on the implementation 52
4SML Demo 53
Some comments on the implementation 53

Client/server use of RealSpeak 54
Intro 54
Running the TTS Server 54
Intro 54
Configuring the server 54
Specifying the installation directory 55
API Call sequence 55
Demonstration applications 56
Twonode Demo 56
Some comments on the implementation 57
Dict_n_rules Demo 57
Some comments on the implementation 59

RealSpeak Parameters 60
Introduction 60
Use of Configuration Files 61
Setting of Parameters via the API 61
Non-speak parameters 61
Speak Parameters 61
Text Markup 62
Overview of RealSpeak parameters 62

Use of RealSpeak in telephone or dialogue applications 68
Multiple engine instances 68
Real-time responsiveness and audio streaming 69

REALSPEAK API 72

New and Changed in RealSpeak 4.0 API 72

Defined Data Types 74
HTTSDICT 74
HTTSDCTEG 74
HTTSINSTANCE 74
HTTSMAP 74
HTTSVECTOR 74

TTSRETVAL 75
LH_SERVER_INFO 75
LH_SDK_SERVER 75
TTSCallBacks 76
TTSPARM 76
TTS_PARAM 79
TTS_PARAM_VALUE_T 80
TTS_PARAM_T 80
TTS_FETCHINFO_T 81
SpeakData (PSpeakData) 82
DictionaryData, (PDictionaryData) 87
G2P_DICTNAME 88
TTS_Marker 89
TTS_Event 89
TTS_MarkPos 91
TTS_BookMark 91
TTS_PhonemeMark 93
TTS_SentenceMark 93
TTS_ParagraphMark 93
TTS_WordMark 94

Function Descriptions 95
TtsInitializeEx 95
TtsInitialize 96
TtsUninitialize 97
TtsProcessEx 98
TtsProcess 99
TtsStop 100
TtsSetParam 101
TtsGetParam 104
TtsSetParams 105
TtsGetParams 106
TtsLoadUsrDictEx 107
TtsLoadUsrDict 108
TtsUnloadUsrDictEx 109
TtsUnloadUsrDict 110
TtsEnableUsrDictEx 111
TtsEnableUsrDict 112
TtsDisableUsrDictEx 113
TtsDisableUsrDict 114
TtsDisableUsrDictsEx 115
TtsLoadG2PDictList 116
TtsUnloadG2PDictList 117
TtsGetG2PDictTotal 118
TtsGetG2PDictList 119
TtsMapCreate 120
TtsMapDestroy 121
TtsMapSetChar 122
TtsMapSetU32 123
TtsMapSetBool 124
TtsMapGetChar 125
TtsMapFreeChar 126
TtsMapGetU32 127
TtsMapGetBool 128
TtsCreateEngine 129
TtsRemoveEngine 130
TtsResourceAllocate 131
TtsResourceFree 132

User Callbacks **133**
TTSSOURCECB 133
TTSDESTCB 135
TTSEVENTCB 136

Error Codes **137**

SAPI5 COMPLIANCE 141

API Support **141**

SAPI5 Interface **143**
ISpVoice Interface 143
I S p V o i c e : :ISpEventSource 144
ISpVoice::SetOutput 144
ISpVoice::GetOutputObjectToken 144
ISpVoice::GetOutputStream 144
ISpVoice::Pause 144
ISpVoice::Resume 144
ISpVoice::SetVoice 144
ISpVoice::GetVoice 144
ISpVoice::Speak 144
ISpVoice::SpeakStream 145
ISpVoice::GetStatus 145
ISpVoice::Skip 145
I S p V o i c e : :SetPriority 145
ISpVoice::GetPriority 145
ISpVoice::SetAlertBoundary 145
ISpVoice::GetAlertBoundary 145
ISpVoice::SetRate 145
ISpVoice::GetRate 145
ISpVoice::SetVolume 146
ISpVoice::GetVolume 146
ISpVoice::WaitUntilDone 146
ISpVoice::SetSyncSpeakTimeout 146
ISpVoice::GetSyncSpeakTimeout 146
ISpVoice::SpeakCompleteEvent 146
ISpVoice::IsUISupported 146
ISpVoice::DisplayUI 146
SAPI5 XML Tags 147
Bookmark 148
Context 149
Emph 150
Lang 151
Partofsp 152
Pitch 153
Pron 154
Rate 155
Silence 156
Spell 157
Voice 158
Volume 160
Load ScanSoft User Dictionaries 160

SAPI5 Client/Server **161**
Required Software 161
Required Hardware 162

Installing SAPI5 Layer 162
Change in the Registry 162
Modifications in the Configuration File 162
Load User Dictionaries 163
Microsoft Lexicon 164
Enable Logging 164

SSML SUPPORT 167

Introduction and Purpose 167
Links 167

SSML Compliance 167
Support for the SSML 1.0 REC September 2004 167
Legacy support for the SSML 1.0 WD December 2002 168
Legacy Support for the SSML 1.0 WD April 2002 169
Volume Scale Conversion 170
Rate Scale Conversion 171
Break Implementation 172
Say-as Support 172
The Lexicon Element 176

Scansoft SSML Extensions 177

API functions 178

LANGUAGE IDENTIFIER 1.0 180

Language Identifier 1.0: Preface 180
Overview 180
System Requirements 180
Size requirements 180
OS requirements 181
Software requirements 181

Installing the Language ID software 182
Installing 182

Using the Language ID software 183
Overview 183
Language set 183
Available Languages and Codings 184
Language Classification 185
Tuning Classification 185

Language ID API Functions 186
Data structure reference 187
LID_H 187
LID_SCORE_T 187
lid_ObjOpen() 189
lid_ObjClose() 190
lid_Identify() 191

USER CONFIGURATION 193

Overview 193

User Dictionaries **193**
Functional Description 193
Dictionary substitution rules 194
Dictionary Format for RS Host version 4.0 195
Dictionary format for older RealSpeak versions (3.x) 199
Migration from 3.x to 4.0 format 200
Case 1: Orthography only 200
Case 2: phonetics only 201
Case 3: both orthography and phonetic 201
User Dictionary API calls 202
Restrictions on user dictionaries 203
Automated User dictionary Loading 203
User Dictionary Editor (Windows only) 204

User Rulesets **205**
Introduction 205
Tuning of text normalization via rulesets 205
Ruleset format 206
Header Section 206
Data Section 207
Rule example 208
Search-spec 208
Replacement-spec 209
Some examples of rules 209
Restrictions on rulesets 210
Effect of rulesets on the TTS performance 210
Ruleset API functions 210
Sample code 211
Automated ruleset loading 212

Custom G2P Dictionaries **213**

Custom Voices **214**

Configuration Files **215**

Configuration file format **215**

Configuration parameters **217**
Single value parameters 217
Environment Variable Overrides 217
Element 217
Description 217
Default 217
Optional 217
Network Parameters 217
Elements 217
Description 217
Default 217
Optional 217
Licensing Parameters 218
Elements 218
Description 218
Default 218
Speak Parameters 219
Elements 219
Description 219
Default 219

Miscellaneous Server Parameters 219
Elements 219
Description 219
Default 219
Internet Fetch Cache Parameters 219
Elements 219
Description 219
Default 219
Internet Fetch Parameters 220
Elements 220
Description 220
Default 220
Diagnostic and Error Logging Parameters 221
Elements 221
Description 221
Default 221
Multiple Value parameters 221
inet_extension_rules 221
default_dictionaries 221
default_rulesets 222
license_servers 222

REALSPEAK E-MAIL PREPROCESSOR 224

Introduction 224

E-Mail Header Processing 225
Header Field Extraction 225
Header Field Reading 226

E-Mail body processing 227
Message Extraction 227
Text Normalization 227

Customizing the E-Mail Preprocessor 227

Support for markup in E-mail mode 227
Native markup 228
SSMLmarkup 228

E-mail Preprocessor API functions 228
Sample code 228

SPEECHIFY API 231

Introduction 231

API Reference 231

Calling convention 231

SDK's preferred character set 232

Result codes 232

SWIttsAddDictionaryEntry( ) 235
Mode 235

Purpose 235
Notes 235

SWIttsCallback( ) 236
Mode 236
Purpose 236
Parameters 236
Structures 238
Notes 240

SWIttsClosePort( ) 242
Mode 242
Purpose 242
Parameters 242
See also 242

SWIttsDeleteDictionaryEntry( ) 243
Mode 243
Purpose 243
Notes 243

SWIttsDictionaryActivate( ) 244
Mode 244
Purpose 244
Parameters 244
See also 245

SWIttsDictionariesDeactivate( ) 246

SWIttsDictionariesDeactivate( ) 246
Mode 246
Purpose 246
Parameters 246
See also 246

SWIttsDictionaryFree( ) 247
Mode 247
Purpose 247
Parameters 247
See also 247

SWIttsDictionaryLoad( ) 248
Mode 248
Purpose 248
Parameters 248
Structures 248
See also 251

SWIttsGetDictionaryKeys( ) 252
Mode 252
Purpose 252
Notes 252

SWIttsGetParameter( ) 253
Mode 253
Purpose 253
Parameters 253
See also 256

SWIttsInit( ) **257**
Mode 257
Purpose 257
Parameters 257
Notes 257
See also 257

SWIttsLookupDictionaryEntry( ) **258**
Mode 258
Purpose 258
Notes 258

SWIttsOpenPort() **259**
Mode 259
Purpose 259
Notes 259
See also 259

SWIttsOpenPortEx( ) **260**
Mode 260
Purpose 260
Parameters 260
Notes 261
Example 262
See also 262

SWIttsPause( ) **263**
Mode 263
Purpose 263
Parameters 263
Notes 263
See also 263

SWIttsPing( ) **264**
Mode 264
Purpose 264
Parameters 264
See also 264

SWIttsResetDictionary( ) **265**
Mode 265
Purpose 265
Notes 265

SWIttsResourceAllocate( ) **266**
Purpose 266
Parameters 266
Notes 266
See also 266

SWIttsResourceFree( ) **267**
Purpose 267
Parameters 267
Notes 267
See also 267

SWIttsResume( ) **268**
Mode 268

Purpose 268
Parameters 268
Notes 268
See also 268

SWIttsSetParameter( ) **269**
Mode 269
Purpose 269
Notes 269
See also 271

SWIttsSpeak( ) **272**
Mode 272
Purpose 272
Parameters 272
Notes 272
See also 273

SWIttsSpeakEx( ) **274**
Mode 274
Purpose 274
Parameters 274
Structures 274
See also 276

SWIttsStop( ) **277**
Mode 277
Purpose 277
Parameters 277
Notes 277
See also 277

SWIttsTerm( ) **278**
Mode 278
Purpose 278
Parameters 278
Notes 278
See also 278

SPEECHIFY EMAIL PRE-PROCESSOR **280**

Introduction **280**
Features 280

Order of API calls **281**

FUNCTIONALITY OF THE E-MAIL PRE-PROCESSOR **283**

In This Paragraph **283**

Supported message formats **284**

Default behavior **285**
Header processing 285
Discarding header lines 285
Reading From lines 286

Subject line abbreviations ... 286
Body processing ... 286
Discarding data ... 286
Multiple punctuation marks ... 287
Embedded e-mail messages ... 287
Signature processing ... 288
MIME format ... 288

Modes ... 289

USING THE E-MAIL SUBSTITUTION DICTIONARY ... 290

In This Paragraph ... 290

File format ... 290

Dictionary entries ... 291

Comments and escapes ... 292

Notifications ... 292

API REFERENCE ... 295

In This Paragraph ... 295

Calling convention ... 295

Result codes ... 296

SWIemailInit( ) ... 297
Mode: Synchronous ... 297
Notes ... 297

SWIemailProcess( ) ... 298
Mode: Synchronous ... 298
Notes ... 298

SWIemailTerm( ) ... 299
Mode: Synchronous ... 299

APPENDICES ... 301

Appendix: TTSPARM member values ... 301

Appendix: RealSpeak API Function Directory ... 303

Appendix: Running a TTS server as a service (Windows only) ... 305

Appendix: Port density simulator ... 306

Appendix: Copyright and Licensing for third party software ... 307
ADAPTIVE Communication Environment (ACE) ... 307
Apache Group ... 309
The Flite Speech Synthesis System ... 310
Dinkumware C++ Library for Visual C++ ... 310

RSA Data Security, Inc. MD5 Message-Digest Algorithm....310
ICU....310
PCRE....311

Appendix: RealSpeak Languages....313

Appendix: Tips for using RealSpeak....315
Operating System Restrictions....315
Optimal Audio Buffer size....315
Limiting delays when internet fetching is used....315
Binary versus textual user dictionaries....316

RealSpeak Telecom
Software Development Kit

Chapter I

Introduction

Programmer's Guide

Chapter I

Introduction

Introduction to RealSpeak

This guide provides operational instructions for the RealSpeak Telecom Text-To-Speech (TTS) system. It reviews the functionality of the system and explains how the various APIs can be used to integrate TTS into an application, and describes the ways in which the user can customize the pronunciation of input texts.

Organization of this manual

The following table shows the organization of this manual:

Chapter I: Introduction describes this guide and the technical support services for the RealSpeak TTS product.
It also explains the architecture of the TTS system and the new features of the RealSpeak v4 release.

Chapter II: Installation explains how to install the SDK and configure the license protection.

Chapter III: Deploying RealSpeak describes the use of the API for various system configurations, illustrated by a discussion of the demonstration applications.

Chapter IV: RealSpeak API contains a detailed explanation of all the API functions, data structures and type definitions.

Chapter V: SAPI5 Compliance describes the support for the Microsoft SAPI5 interface.

Chapter VI: SSML Support describes the support for the XML-based SSML v1.0 markup language.
RealSpeak extends SSML with a number of Scansoft specific elements/attributes. The set supported by Scansoft is called "ScanSoft SSML" (4SML).

Chapter VII: Language Identifier describes the language identifier component and his API.

Chapter VIII: User Configuration describes the different ways in which a user can tune RealSpeak. It describes User Dictionaries, User

Chapter I

Rulesets, Custom G2P dictionaries, Custom Voices and Configuration Files.

Chapter IX: RealSpeak Email Pre-Processor

Chapter X: Speechify API describes the support for the SWItts API of Speechify 3.0. This support eases the migration of existing Speechify based integrations and applications to the next-generation RealSpeak products that incorporate Speechify technology. New software should only be developed using the native RealSpeak APIs or the Microsoft SAPI 5 APIs, however.

Chapter XI: Speechify Email Pre-Processor

The appendices provide additional information for using the SDK. They cover such topics as "RealSpeak API Function Reference", "Running a TTS Server as a Service", "RealSpeak Languages" etc.

Contacting ScanSoft

ScanSoft wants you to get the most from its software. To receive technical support from ScanSoft, Inc., visit http://developer.scansoft.com.
This site requires a customer username and password.
You can also visit http://www.scansoft.com for general corporate, product, marketing, and sales information.

Defect Report Form

If you believe that you have found a defect in the RealSpeak Telecom software, please contact technical support using the contact information provided above. In reporting the problem you must supply all of the information described in the defect report form, which is provided as a plain text file named DEFECT_REPORT_FORM on the product CD. The easiest way to use the form is to copy the text from the form into an email and send it to technical support along with any required attachments.

System Overview

Introduction

This section describes:

- API Support

Chapter I

- Markup Support
- Product Support
- New features of RealSpeak v4
- The input/output behavior of RealSpeak: the different input mechanisms and audio streaming
- The modes of operation: in-process and Client/Server
- Use of RealSpeak in telephony environments

API Support

RealSpeak Telecom V4.0 supports the following API's:

- New RealSpeak Telecom 4.0 API
- Old RealSpeak Telecom 3.51 API
- Microsoft SAPI 5
- Speechify 3.0 API: RealSpeak Telecom almost fully supports the SWItts API of Speechify 3.0. This support eases the migration of existing Speechify based integrations and applications to the next-generation RealSpeak products that incorporate Speechify technology. New software should only be developed using the native RealSpeak APIs or the Microsoft SAPI 5 APIs, however.

Markup Support

The input text can be marked up to control aspects of the generated speech such as voice, pronunciation, volume, rate, etc.
RealSpeak supports several markup languages:

- The RealSpeak native markup language which is explained in the language specific User's Guide of each RealSpeak language.
- W3C SSML v1.0 (XML-based) with some proprietary extensions, called 4SML. This is described in the "SSML Support" chapter.
- SAPI v5 XML tags. The support for SAPI is described in the "SAPI5 compliance" chapter.

The RealSpeak API supports two markup languages: the **native** one and 4SML (**SSML** with some proprietary extensions). The same markup languages are supported when the Speechify API is used. **SAPI5** tags are supported when the SAPI interface is used.

Product Support

RealSpeak Telecom V4.0 can be used with the following SpeechWorks Solutions from ScanSoft:

- SWMS 3.1
  RealSpeak Telecom V4.0 has been used for integration in our MRCP product line, SWMS 3.1. For more information please read the SWMS 3.1 documentation or visit http://developer.scansoft.com

Input/Output behavior of RealSpeak

Three different text input techniques

The input text can be provided by the application in three ways: as an input stream, a document specified via a URI or a text buffer.
The **input stream method** relies on the use of the TTS source call-back function, implemented by the application. This function is called by the RealSpeak engine when it needs to receive a next block of input text. Figure I-1 illustrates this mode of input. RealSpeak v3.5 supported only this input technique.
The input can also be specified as a document specified via a **URI.** RealSpeak supports documents on an HTTP server and local files. Figure I-2 illustrates this mode of input. This method is particularly useful in Client/Server configurations with multiple servers: the input texts can be stored on a central Web server. RealSpeak supports caching of the retrieved documents and use of a proxy server.
The third input way is to provide a **text buffer** when the TTS Process function is called.

Presentation of the input text

The input text can be marked up to control aspects of the generated speech such as voice, pronunciation, volume, rate, etc.
As already described under the "Markup Support" section above, RealSpeak supports three **markup languages**:

- The RealSpeak native markup language
- W3C SSML v1.0 with some proprietary extensions, called 4SML
- SAPI v5 XML tags

RealSpeak supports a wide range of **character sets** and **encodings**. The engine handles the transcoding of the input text to the native (or internal) character set of the active language.

Language and voice switching

The active language and voice can be specified when a TTS engine instance is initialized, in-between the initialization and the TTS request, or during the processing of input text (via the markup). The voice and language can be switched at any location in the input text but will result in a sentence break.

Audio output streaming

Chapter I

The audio output is streamed to the application via the Destination call-back. This is a handler implemented by the application which receives the audio chunk by chunk. The application can specify the desired audio format (A-law, mu-law, 16-bit linear etc).

Modes of operation: in-process and Client/Server

ScanSoft RealSpeak Telecom can operate in client/server and in-process (or single process) mode.

In-process mode

For in-process mode the RealSpeak service is fully implemented by libraries (primarily DLL's or shared objects) linked in by the user's application. All TTS components are then running in the same process, so there is no communication overhead.
Figure I-1 shows an example system layout of an application using RealSpeak in-process. Only one RealSpeak voice, being American English Jill, has been installed on the machine. The application has created one RealSpeak engine instance via the RealSpeak API. The figure shows that the application has chosen to provide the text input via the TTS source call-back. In that case the input text is streamed to the RealSpeak engine instance. The audio streams in the opposite direction from the engine instance back to the application.

Chapter I

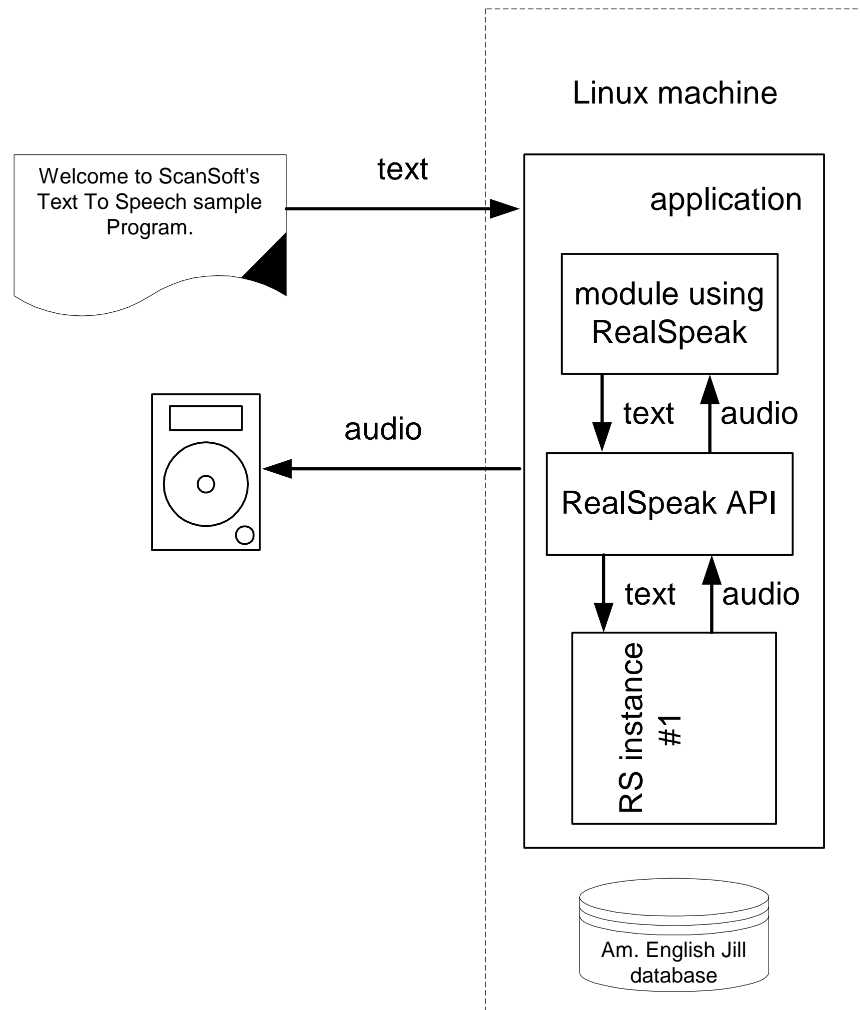


Figure I-1

When the application specifies the text input for a Speak request via a URI, the architecture looks like Figure I-2. In that case, the RealSpeak instance will rely on the Scansoft internet fetch component to retrieve the content of the URI. Note that this is a simplified presentation of the internet fetching: in reality the fetch library uses a configurable cache, so the overhead of retrieving previously downloaded documents is minimal.

Chapter I

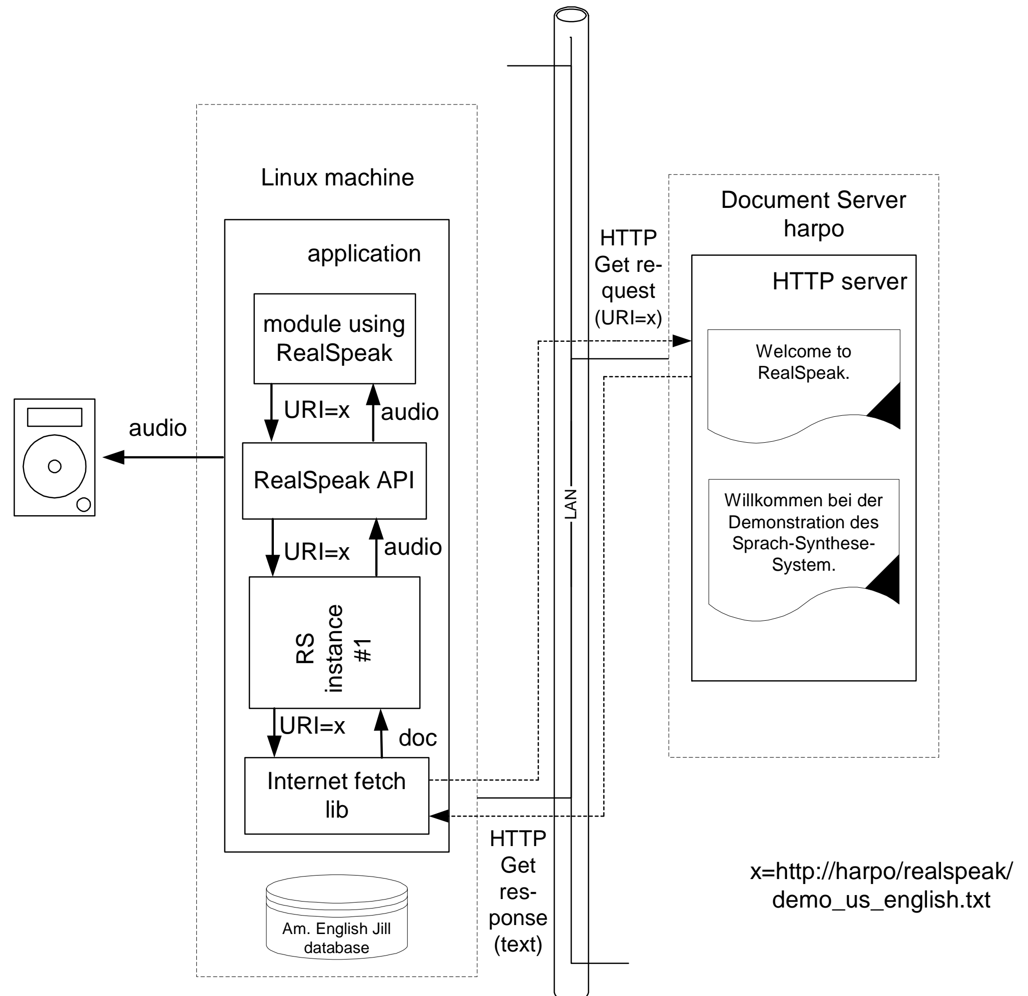


Figure I-2

Client-Server mode

In client/server mode the lightweight RealSpeak client is integrated into the user's application that needs TTS services. The client implements the RealSpeak API layer and each RealSpeak client instance created via the API can be used to send TTS requests to a certain RealSpeak server.
The RealSpeak server (also called TTS server) is a standalone executable that can be started from the command line or, on Windows, as a Windows service. The server can reside on any machine on the network. and clients and servers do not have to run on the same operating system. A server instance is created when a client instance is created and they are connected for their entire life span. A server instance performs the actual TTS conversion and

sends the generated speech to the corresponding client instance. The client instance then passes it back to the application.
The TTS Server can create multiple server instances and each instance runs in a separate thread; so each RealSpeak server can handle multiple requests (one per instance) simultaneously.
A TTS server instance can handle TTS requests for all the RealSpeak languages and voices that have been installed on the server machine. And there is no hard limit on the number of languages and voices that can be installed on a single server machine.
The client instance tells the server which initial language, accent and/or voice to use for the next TTS request. Note that the input text for the TTS request can contain markup to switch the language and/or voice.
Multiple TTS server processes (or machines) can work with multiple clients (or applications).
RealSpeak is fully multi-threaded, and so can run on multiple CPU machines very efficiently.
Figure I-3 shows a snapshot of an example system layout with a client on a Linux machine that is connected with one of two available servers: one running on a Windows and one on a Linux machine. The example Linux server has the availability of two RealSpeak voices: American English "Jill" voice and British English "Emily" voice. The figure shows a snapshot of the situation at time x. Before that time, the application has used the RealSpeak API to create a RealSpeak server engine instance #1 on the Linux server and a client instance #1 on the application machine connected with the server instance. Note that RealSpeak only allows one client instance to be connected with one server engine instance for the life span of the client instance.
After the successful initialization of the server and client instance, the application has requested the client instance to convert an American English text to speech.
Note that the active language and voice can be specified when the engine instance is initialized, in-between the initialization and the TTS request, or during the processing of input text (via the markup).
Figure I-3 shows a snapshot while the server engine instance is performing TTS for an American English document. The figure shows that the application has chosen to provide the text input via the TTS source call-back. In that case the input text is streamed to the server engine instance via the client instance. The audio streams in the opposite direction from the server engine instance back to the application via the client instance.

Figure I-3

When at a later point in time, a German text needs to be processed, the application must create an engine instance on the Windows server since the Linux server has not been equipped with a German voice.

Chapter I

The application has the option to keep client instance #1 and corresponding server engine instance #1 for TTS processing of future American or British English texts. In any case, a new server engine instance #2 has to be created on the Windows server; followed by the creation of a client engine instance which is connected to that server engine instance. Figure I-4 is a snapshot of the processing of the German text.

Chapter I

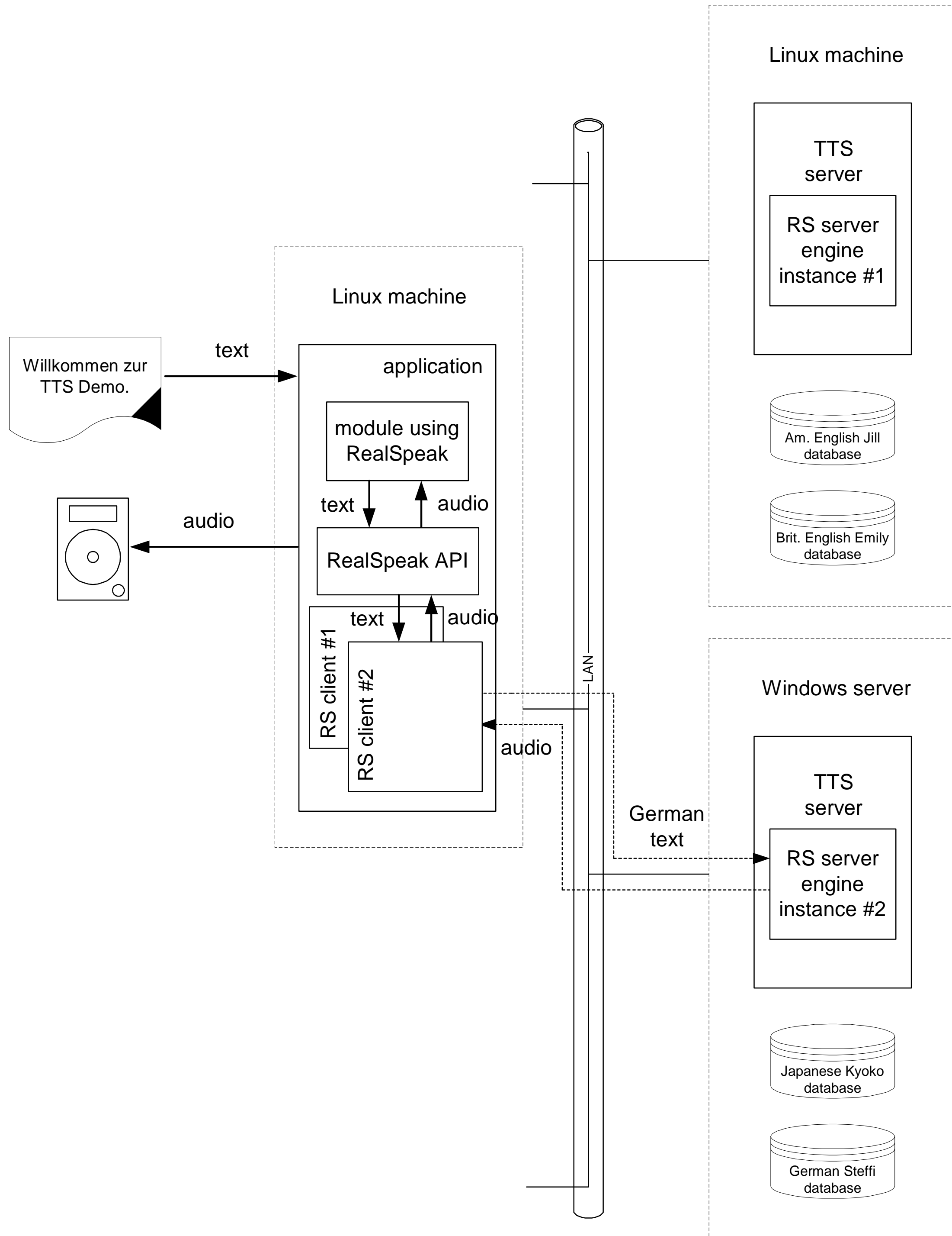


Figure I-4

Chapter I

Use of RealSpeak in telephony environments

There are three components to any telephony application using speech technology:

- The main application
- The Voice Source
- The Engine

The main application is the brains of the entire system and is responsible for the overall setup and control of the speech engines and Voice Sources. The main application is the master of the system; the Voice Source and the Engine are slaves to the main application. The Voice Source is the point at which voice input or output occurs in the telephony application. In traditional telephony applications, the point of entry is the telephony voice board, and the Engine is the speech Engine (Text-To-Speech and/or Speech Recognition). This Engine can be running on a different machine.
Telephony applications are designed to service many customers at the same time. The concept of "voice port" is often used in this domain. Each voice port can service one customer at a time. One port is usually associated with one telephone line.

There are two major system models, loosely relating to the number of nodes (computers) that these pieces are running on:

- Single Node – In a single node system, or in-process model, all components are running on the same computer. This is the typical configuration in small systems handling a small number of lines. In the single node system, all voice data can be routed between the Voice Source and the Engine through the main application with no network overhead. Figure I-1 and I-2 closely match this configuration. The application will stream the audio output by the RealSpeak engine to the Voice Source. Typically, one RealSpeak engine instance will serve one port.
- Two Node – In a two-node system, or client-server model, the main application and the Voice Source are located on one computer and the Engine is located on another. This configuration allows the application to offload the heavy Engine processing, allowing the main application to handle more ports on a single machine. Separating the Engine creates a modular system that is more fault-tolerant, flexible, and manageable. In this system, the application can still be the middleman streaming all voice data between the Engine and the Voice Source.

Chapter I

New features for RealSpeak 4.0

The functionality of the TTS system has been improved compared to the previous release (v3.5x). The ScanSoft RealSpeak SDK has the following new and changed features for V4.0:

- Support for new RealSpeak v4.0 API while maintaining support for RealSpeak v3.5 API
- SAPI 5 and Speechify v3 API support come as an integral part of the SDK
- New Improved User Dictionary Editor. Windows only. Allows saving to other platform types
- Improved SSML support
  - Support for <audio> element to allow for easy insertion of prerecorded audio files.
  - Support for SSML 1.0 Recommendation of September-2004
- Language identification module has been added
- Extended API functions that support new methods for specifying the input of the Speak. Before, the TTS source call-back (input streaming) was the only input method.
  - Text input can now also be specified via a URI with optional fetch properties and cookie jar. Fetch properties can be especially handy in case of remote input data (e.g. for specifying a fetch timeout).
  - Input can also be provided via a text buffer.
  - When one of two above input methods is used, the content type and character set of the input can be specified. The content type specifies the document type: "standard" (use of native markup) versus 4SML (or SSML).
    This version supports a wide range of character encodings; RealSpeak now handles the transcoding of the input text to the native (or internal) character set of the active language.
- Extended dictionary loading functionality, accessible via new API function
  - Load from URI with optional content-type, fetch properties and cookie jar
  - Load from a memory block

- Support for multiple active user dictionaries with user specified priorities
  - With the new API functions dictionaries are specified per TTS instance
- Introduction of license protection: a floating licensing scheme is in use

Chapter 1

- Extending the number of parameters that can be changed via the API once a TTS instance is created: sample frequency, language, voice, etc. More than one parameter can be switched at a time.
- Support for the signaling of the following types of markers via the Event callback: user bookmarks, paragraph[1], sentence, word and phoneme markers

[1] Paragraph markers are only signaled when paragraphs have been marked in the input text via the paragraph tag (native <ESC>\p\ tag or SSML <p> element).

RealSpeak Telecom
Software Development Kit

Chapter II

Installation Guide

Chapter II

Installation Guide

Licensing

Licensing - Important note

RealSpeak Telecom requires a valid license file to perform TTS services. This license file is NOT supplied with the software but can be obtained from Scansoft.

Overview of licensing

RealSpeak Telecom uses run-time licensing that is based on the number of initialized TTS engine instances, and is co-resident with LAN based licensing servers. RealSpeak uses the "Flex License Manager" (FLEXlm) third-party software to implement a **floating license model**, so licenses are not required to be dedicated to a specific RealSpeak server (or RealSpeak machine for in-process mode). Instead, one license is needed per active TTS instance, regardless of the machine it is running on. There are two floating licensing modes: implicit (the default) and explicit.

Implicit licensing means that licenses are acquired automatically when a TTS engine instance is created via the TtsInitialize(Ex) API function and released when the instance is destroyed via TtsUninitialize.

To support **explicit licensing**, the functions TtsResourceAllocate and TtsResourceFree were added to the API. These functions enable the developer to choose when he wants to acquire and release licenses.

To configure the licensing to explicit or to implicit mode, an environment variable can be set. The values for this variable (SSFT_TTS_LICENSE_MODE) can be 'default' or 'explicit' where 'default' means implicit mode. When the environment variable is not defined, the default or implicit mode will be used.
In client/server mode, licenses are acquired by the TTS server and the default license mode can then be set via the license_mode parameter in the server configuration file.
But the application can override the default setting via TTS_LICENSE_MODE_PARAM parameter which can be set when a TTS engine instance is created, or via SetParam(s).
An application can determine the mode by calling TtsGetParam using TTS_LICENSE_MODE_PARAM as name for the parameter. See

Chapter II

the TtsSetParam entry in the "RealSpeak API" chapter for the possible values of this parameter.

The FLEXlm components are part of the RealSpeak Telecom installation package.
A FLEXlm license server must be set up on a machine on the same logical subnet as the RealSpeak servers or machines in case of in-process RealSpeak. The document "./doc/RS_Telecom_TTS_Licensing_Handbook.pdf" under the Realspeak installation directory describes the licensing in great detail. It explains how to obtain and manage licenses and how to install and configure a license server. Note that this document refers to the in-process mode of RealSpeak as "all-in-one".
Some Windows and Linux specific details are also described in the following Installation sections.

Installation on Windows

Realspeak for Windows can be run on Windows 2000, Windows XP Professional (client or in-process mode only) and Windows Server 2003.
The installation of RealSpeak Telecom 4.0 is a very straightforward process that consists out of two major steps: common install and voice specific install. Both installers consist of a limited number of input screens which makes it possible for the user to provide the necessary information and to configure the SDK.
The common installer gives the user the opportunity to:

- Configure the install directory
- Install the license server
- Install RealSpeak Telecom as a windows service.

The voice specific installer uses the information retrieved during the installation of the common part to determine the target path of the voice specific installer.

Installation Steps for Windows

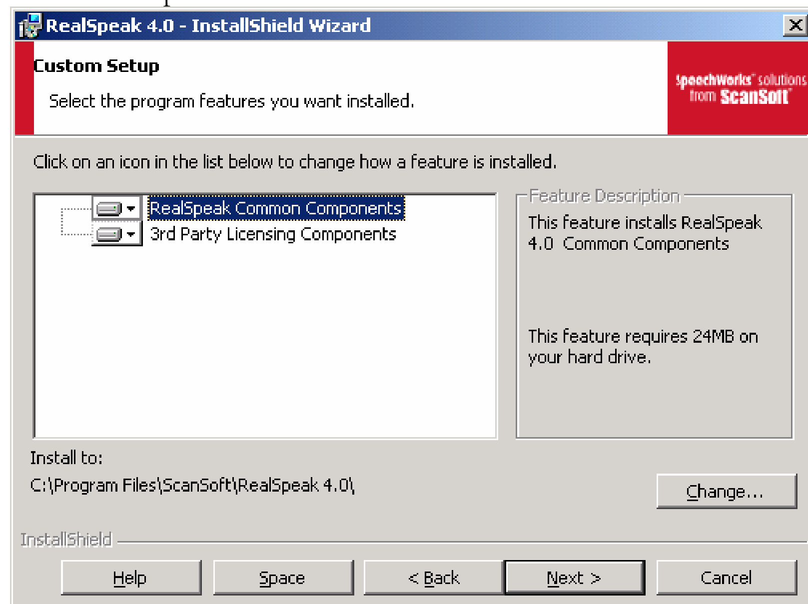
Install the common installer

'Startup' screen

Chapter II

Click 'Next' to continue.

'Custom Setup' screen

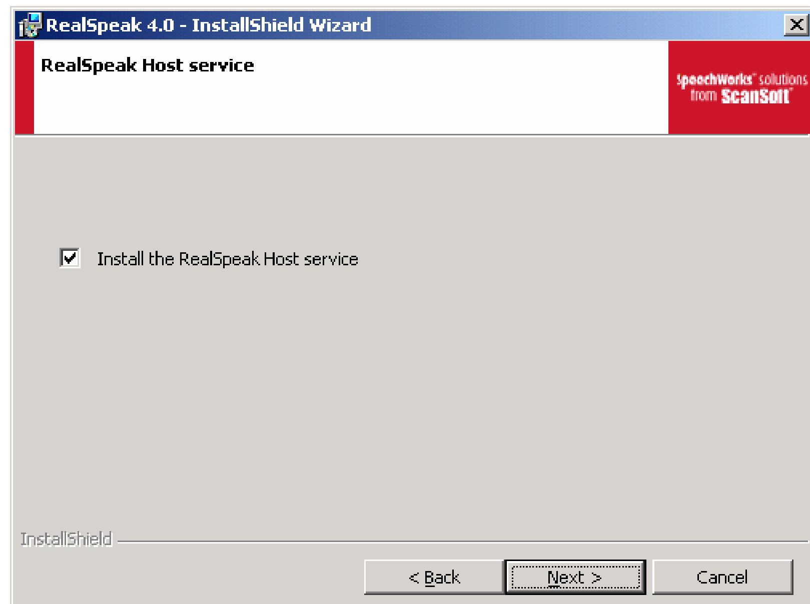


Here you can specify which components you want to install. You can install the common part and the third party licensing software (FLEXlm). Next to that you can also specify the installation path

Click 'Next' to continue.

'Host service' screen.

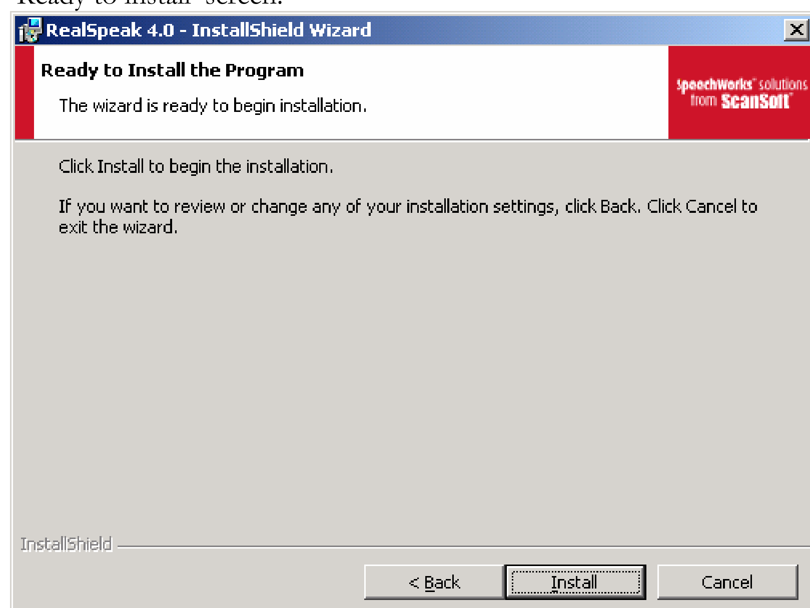
Chapter II



Here you can specify whether you want the TTS server installed as Windows Service or not. Uncheck the box if you do not want the service installed. When checked, the TTS service will be installed with startup type 'automatic' (starts automatically when the Windows system starts). Note that the service's options can be changed later via the "Windows Control Panel" by selecting "Administrative Tools/Services/RealSpeak Host".

Click 'Next' to continue

'Ready to install' screen.



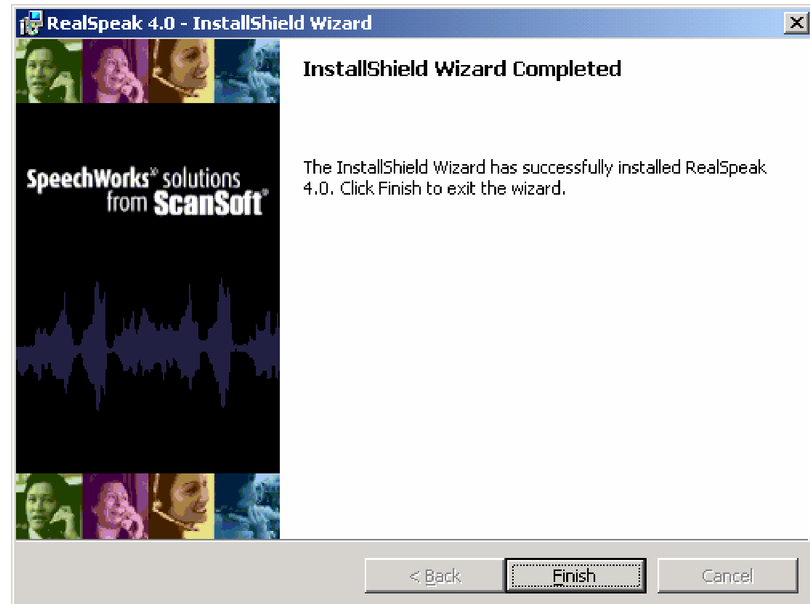
The setup is now ready to install the selected components.

Chapter II

Click 'Install' to continue.

The installer will now install all the selected components.

'Finish' screen.



Click 'Finish' to complete the installation.

Install the voice specific installer

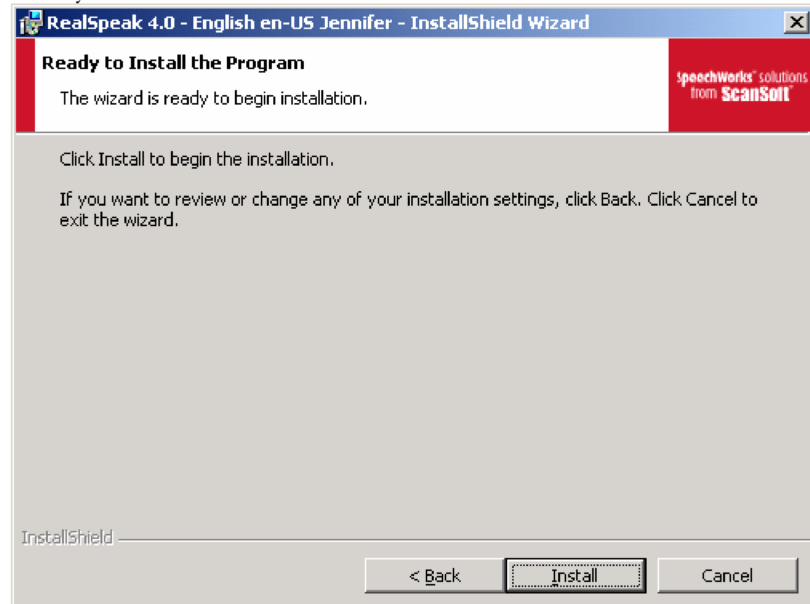
Repeat the following for every voice that needs to be installed.

Chapter II

'Startup' screen

Click 'Next' to continue.

'Ready for installation' screen.



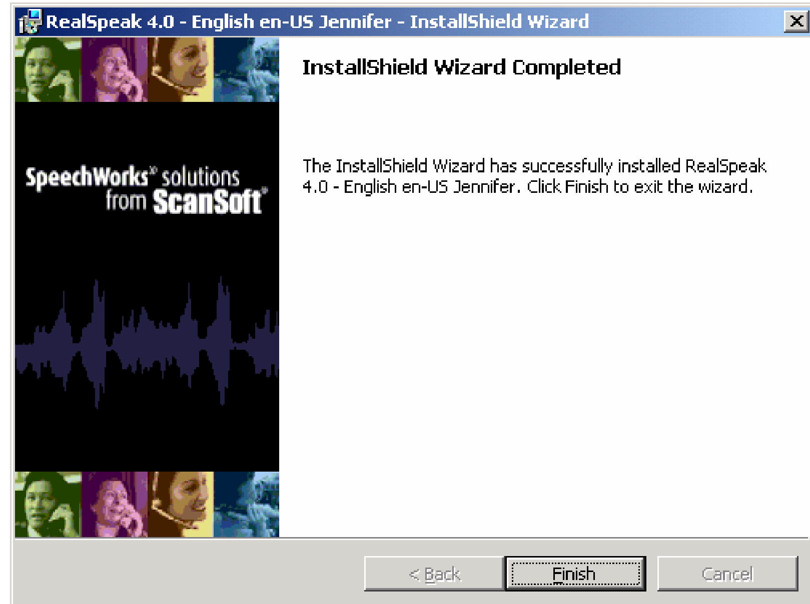
The setup is now ready to install the selected components.

Click 'Install' to continue.

The installer will now install the selected voice.

Chapter II

'Finish' screen.



Click 'Finish' to complete the installation

Configuring the licensing

Please consult the document "./doc/RS_Telecom_TTS_Licensing_Handbook.pdf" under the Realspeak installation directory for detailed information on licensing. In a nutshell, here's the procedure for installing a FLEXlm license server on Windows. The license server can also be installed on a Linux machine.

a) Determine the hostid of the Windows license server
   In order to generate a license, Scansoft requires the hostid (Ethernet address for Windows) of the machine that will run the license server. The hostid can be obtained as follows:
   - change to the directory %SSFTTTSSDK%\flexlm\components
   - and run:
     lmutil.exe lmhostid
b) Use the returned hexadecimal digit string to obtain your license file from the Scansoft and place it on your system. Consult the Licensing Handbook, section "Obtaining and managing licenses, subsection "Generating licenses and downloading license files" for the details.
c) Install and Configure the license server
   By default, the license software has already been installed when installing RealSpeak on a Windows machine.
   Also by default the installation will configure the "RealSpeak

Licensing Service” to start automatically with every system reboot. Consult the Licensing Handbook, section “Configuring and starting the license server” for the details.

d) Configure the RealSpeak server to use the appropriate license server
Consult the Licensing Handbook, chapter “Configuring Licensing on Windows”, section “Configuring RealSpeak on Windows” for the details. When using RealSpeak in client/server mode this includes updating the RealSpeak server configuration file variables license_mode and license_servers.
When using RealSpeak in in-process mode, the only option is to set the environment variables SSFT_TTS_LICENSE_MODE and SSFT_TTS_LICENSE_SERVERS to appropriate values.

Running a demo program

For demonstration purposes, RealSpeak comes with several applications. The simplest one is the “standard” program. This can be run to do a quick verification of the installation. It runs RealSpeak in-process. It processes one text file which can contain RealSpeak native markup for the specified language. The output is one linear 16-bit PCM speech file (with 8kHz sample rate) named “standard.pcm”. It is explained in more detail in Chapter III, section “In-process use of RealSpeak”, subsection “Demonstration applications”.

Instructions

- Go to the RealSpeak installation directory, e.g. "C:\ program files\ScanSoft\RealSpeak 4.0" and open a command prompt.
- It should be run as follows:
  standard <language> <voice> <engine directory> <text file>
  Running the program without arguments shows a help screen)
  e.g.
  standard.exe "American English" Jennifer .\speech .\api\demos\data\us_english.txt
- If the program returns with error 120, it means that TTS could not acquire a license. You should check your license configuration.
- If no error is returned everything went fine and a PCM file called standard.pcm is generated.

Chapter II

Installation on Linux

The RealSpeak SDK for Linux RedHat 7.1, 7.2 and 9.0, Linux RedHat Advanced Server 2.1 and 3.0, Linux RedHat Enterprise WS 3.0, Linux RedHat Enterprise ES 3.0.

RealSpeak Telecom is distributed in RPM format.

The "API install CD" comes with two RPM files, one for the RealSpeak common components (including the API libraries) and one for the licensing components.
A "voice install CD" comes with one RPM file which is voice specific.
The default location of the install is /usr/local/ScanSoft/RealSpeak_4.0.
You need to be root or have su permissions to install the software.
The API RPM must be installed first. The RPM's are relocateable.

NOTE: If you do relocate the common components RPM, be sure to relocate any subsequent voice RPM to the same directory.
Please see the RPM man pages for additional options.

Installation Steps for Linux

Step 1: Install the common components

Insert the API install CD into the CD drive and mount the drive.

```
rpm -i rs-api-4.0-0.i386.rpm
```

This will install the RealSpeak common components to the default directory /usr/local/ScanSoft/RealSpeak_4.0

Step 2: Install the purchased voices

```
rpm -i rs-<full-voice-spec> -4.0-0.i386.rpm
```

where <full-voice-spec> is the full specification of the voice to be installed, e.g. American-English-en-US-Jennifer.

Step 3: Install the licensing components

```
rpm -i rs-lic-4.0-0.i386.rpm
```

Step 4: Configuring and starting of the licensing

Please consult /usr/local/ScanSoft/RealSpeak_4.0/doc/RS_Telecom_TTS_Licensing_Handbook.pdf for detailed information on licensing.

Chapter II

In a nutshell, here's the procedure for installing a license server on Linux. The license server can also be installed on a Windows machine.

e) Determine the hostid of the Linux license server
Before TTS can be performed, a RealSpeak engine instance requires a license. In order to generate a license Scansoft requires the hostid (Ethernet address for Linux) of the machine that will run the Flex License Manager. The hostid can be obtained as follows:
  - open a terminal and run:
    /sbin/ifconfig eth0
  - from the output copy the numerical string following HWaddr, e.g.,
    00:06:5B:84:28:00
  - remove the colons, e.g.,
    00065B842800

f) Use this number to obtain your license file from Scansoft. Consult the Licensing Handbook, section “Generating licenses and downloading license files” for the details.

g) Start the license server
  - -once you’ve received the license file, rename it to “realspeak.lic” and copy it to
    /usr/local/ScanSoft/RealSpeak_4.0/flexlm/components
  - to manually start the license manager run:
    ./lmgrd -c realspeak.lic
  - to stop the license manger, open a new terminal and run:
    ./lmutil lmdown -c realspeak.lic
  - there is also the possibility to launch the license server automatically (see Licensing Handbook)

Step 5: Updating your environment settings

- Set the SSFTTTSSDK variable to point to the RealSpeak install directory, the default location is '/usr/local/ScanSoft/RealSpeak_4.0'.
  e.g. when using the C shell:
  setenv SSFTTTSSDK /usr/local/ScanSoft/RealSpeak_4.0
- Update the PATH environment variable to include: $SSFTTTSSDK/speech/components/common and /api/lib
  e.g. when using the C shell:
  % setenv PATH
  $SSFTTTSSDK/speech/components/common:$PATH
  % setenv PATH
  /usr/local/ScanSoft/RealSpeak_4.0/api/lib:$PATH
- Update the LD_LIBRARY_PATH environment variable to include:
  /usr/local/ScanSoft/RealSpeak_4.0/speech/components/common and

/usr/local/ScanSoft/RealSpeak_4.0/api/lib
e.g.
% setenv LD_LIBRARY_PATH
/usr/local/ScanSoft/RealSpeak_4.0/speech/components/common:$LD_LIBRARY_PATH
% setenv LD_LIBRARY_PATH
/usr/local/ScanSoft/RealSpeak_4.0/api/lib/:$LD_LIBRARY_PATH

- On Redhat AS/ES 3.0 or Redhat AS/ES 4.0 you MUST set the environment variable LD_ASSUME_KERNEL to the value 2.4.19. It's imperative to choose this value on the mentioned Redhat versions.
  e.g.
  setenv LD_ASSUME_KERNEL 2.4.19
- Configure the RealSpeak server to use the appropriate license server(s)
  When using RealSpeak in in-process mode, set the environment variables SSFT_TTS_LICENSE_MODE and SSFT_TTS_LICENSE_SERVERS to appropriate values.
  Note that when using RealSpeak in client/server mode the licensing is configured via the license_mode and license_servers parameters in the RealSpeak server configuration file.
  Consult the Licensing Handbook, chapter "Configuring Licensing on Linux", section "Configuring RealSpeak on Linux" for the details.

Step 6: Running a sample program

For demonstration purposes, RealSpeak comes with several applications. The simplest one is the "standard" program. This can be run to do a quick verification of the installation.
It runs RealSpeak in-process. It processes one text file which can contain RealSpeak native markup for the specified language. The output is one linear 16-bit PCM speech file (with 8kHz sample rate) named "standard.pcm". It is explained in more detail in Chapter III, section "In-process use of RealSpeak", subsection "Demonstration applications".
Instructions

- Open a terminal and change to the RealSpeak installation directory, e.g. /usr/local/ScanSoft/RealSpeak_4.0
- Make sure you have 'write' access in this dir (you must have root privileges):
  % chmod +w .
- Give the demo program executable rights (or any other demo program) :
  % chmod +x standard
- Run the standard program as follows:
  standard <language> <voice> <engine directory> <text

file>
Running the program without arguments shows a help screen)
e.g.
% ./standard "American English" Jennifer ./speech ./api/demos/data/us_english.txt

- If the program returns with error 120, it means that TTS could not acquire a license. You should check your license configuration.
- If no error is returned everything went fine and a PCM file called standard.pcm is generated.

Environment variables

Before running RealSpeak the following environment variables should be set whatever operating system is used. Some variables are optional, some are only needed on systems running the TTS server or running TTS in-process. Unix specific environment variables are described in the “Installation on Unix” section, subsection “Installation Steps for Linux”.

Name	when	Comments
SSFTTTSSDK	always, optional	RealSpeak install directory, on Windows the default location is "C:\program files\ScanSoft\RealSpeak 4.0". On Unix it is '/usr/local/ScanSoft/RealSpeak_4.0'. When using the TTS server, it can be overwritten in the server configuration file. It can also be specified when a TTS engine instance is created.
PATH	always, required	Add $SSFTTTSSDK/speech/components/common to the PATH.
TTS_LICENSE_MODE	in-process, optional	License mode Possible values: default or explicit. Default value: default Value ‘default’ means implicit licensing. Note that when using RealSpeak in client/server mode the license mode is configured via the license_mode parameter in the RealSpeak server configuration file. See the Licensing Handbook for more details.
SSFT_TTS_LICENSE_SERVERS	in-process, optional	Port number and hostname of one or more license server machines. Value: semi-colon separated list of <port-number> @<hostname> Default value: 27000@localhost

Chapter II

		Note that when using RealSpeak in client/server mode the list of license servers is configured via the license_servers parameter in the RealSpeak server configuration file. See the Licensing Handbook for more details.

RealSpeak Components

The RealSpeak System is made up of a number of components. This section gives a brief overview of those components. All components are installed in the RealSpeak installation directory or one of its subdirectories. Since the installation directory is usually specified via the SSFTTTSSDK environment variable, this variable is used in the following text when specifying path names.

RealSpeak API library

The RealSpeak interface to the TTS system is implemented as a shared object or DLL (with accompanying import library), depending on the platform.

For example, on Windows, the RealSpeak API library is named lhstts.dll (with corresponding import library lhstts.lib) and on Unix lhstts.so.
This is the only library the application is required to link in or explicitly load in order to use the TTS functionality.

In Client/Server mode, the API library communicates with the TTS Server, which in turn loads the underlying TTS engine library, which with the help of the language libraries, does the actual TTS conversion. In in-process mode, the API library loads the TTS engine library directly.

Note that RealSpeak v4 comes with a number of additional API's: SAPI5 API (see "SAPI5 Compliance" chapter) and Speechify API (see "Speechify API" chapter).

TTS API support libraries

The TTS API uses several shared support libraries, like for example the Internet fetching library used to retrieve documents on an HTTP server. These libraries are DLLs for Windows and shared objects for most Unix platforms.
These libraries reside in the "./speech/components/common" subdirectory of the RealSpeak installation directory. On Windows, this subdirectory is automatically added to the path environment variable by the RealSpeak common installer.

Chapter II

The libraries are used by the TTS API, the TTS engine, the language libraries and the TTS Server (see below).

TTS server

The TTS Server is a standalone executable that provides TTS services to applications using the TTS Client/Server mode of the API. The TTS Server executable is called ttsserver.exe on Windows, and ttsserver on all other operating systems. The RealSpeak Telecom installer installs it in the top-level of the RealSpeak Telecom installation directory.

The server can reside in any location on any machine on the network. The TTS Server depends on certain shared support libraries (see the previous section).

Engine and language libraries

The RealSpeak TTS engine and language libraries are implemented as shared object files or DLLs, depending on the platform. Both the TTS API library and the TTS Server dynamically load the TTS engine library which in turn loads the language libraries when they are needed to execute TTS requests.

Demo programs

The SDK includes a number of demo programs. Detailed instructions on how to run them are provided in the "Deploying RealSpeak" chapter and the Product Release Notes for your particular version.

RealSpeak Telecom
Software Development Kit

Chapter III

Deploying RealSpeak

Programmer's Guide

Chapter III

Deploying RealSpeak

Introduction

This chapter explains how to deploy the RealSpeak system.
RealSpeak can be operated in a number of different ways, each suiting a particular type of deployment. First the in-process operation mode of RealSpeak is described. Then the client/server mode is explained.
Although the RealSpeak client(s) and server(s) can be executed on the same computer, greater efficiency can often be achieved when the client(s) and server(s) are executed by different computers connected via a network.
When performing Text-To-Speech a number of parameters can or must be set. The "RealSpeak Parameters" section of this chapter explains the different classes of parameters and how and when they can be set.
The final section "Use of RealSpeak in telephone or dialogue applications" describes the typical, often more complex use of RealSpeak Telecom in telephony environments.

Chapter III

In-process use of RealSpeak

Intro

The simplest configuration comprises of an "in-process" use of RealSpeak. This has already been explained in the "System Overview" section of the Introduction chapter and figures I-1 and I-2 illustrated this configuration.
The application designer links the TTS API Library to the application requiring TTS. The TTS server standalone program is not used at all; all Realspeak libraries will be directly or indirectly loaded into the application.

For demonstration purposes, RealSpeak comes with several applications demonstrating this configuration; these are described below, but first some typical API call sequences are shown.

API Call Sequence

The following call sequence shows how to use the RealSpeak v4 API when operating RealSpeak in in-process mode. See the sample program "standardex", described below, for extra details and a demonstration of the new possibilities.
The 'Application' component refers to the source code that uses the API.
This call sequence is using the default, implicit licensing mode.

1. The Application calls the **TtsInitializeEx** function to create a TTS engine instance. At that time a number of general parameters can be specified such as the default language and voice, the destination call-back used to stream back the audio to the application, the desired audio format. The source call-back function pointer can be specified to use input streaming for the input text. But this is optional since the TTS input can now also be specified at the time a TTS action is requested via the ProcessEx function. This new approach enables the specification of the input via a URI, a filename or a text buffer. When operating RealSpeak in in-process mode, the application must specify the path of the engine library using the *szLibLocation* member of the TTSPARM structure passed into the TtsInitialize function.
2. When using dictionaries, the application defines a DictionaryData structure for each dictionary instance; this structure describes where to find the dictionary data and how to use it. This new approach enables references to URI addresses and supports a number of document types. Fetch properties can be defined in case of remote access to the

dictionary data The Application calls **TtsMapCreate** to create a map data structure in which all fetch properties can be defined. The application calls **TtsMapGe**t and **TtsMapSet** functions to define these properties in the created map data structure.

3. When using dictionaries, the Application calls the **TtsLoadUsrDictEx** function for each dictionary; this function returns a handle to a dictionary instance.
4. Call **TtsSetParams** to specify the initial speak parameters such as language, voice, volume etc.
5. If the old input call-back method is not used, the Application sets up a **SpeakData** structure describing the input text; this structure supports specifying a URI, a filename or a memory block. If fetch properties need to be specified, the Application calls **TtsMapCreate** to create a fetch property map. The application then uses the **TtsMapSetXXX** functions to add properties to the map one by one.
   If the old method of streaming the input via the TTS source call-back is used, the SpeakData structure members specifying the input, being *uri* and *data*, must be set to NULL.
6. The Application calls the **TtsProcessEx** function to convert the input text to audio of the type defined in TTSPARM (specified in step 1). The input can be specified via the SpeakData structure (URI or text buffer) or the old source call-back method. The Process function executes the TTS action synchronously; it only returns when all the speech samples have been generated.
7. The audio is streamed back to the application via the **TtsDestCb** Destination callback.
8. If required, the Application can perform several **TtsProcessEx** calls, with different SpeakData and/or different DictionaryData instances. When using dictionaries, **TtsEnableUsrDictEx**, **TtsDisableUsrDictEx** and **TtsDisableUsrDictsEx** can be used to enable dictionaries, changes the priorities in which dictionaries are called and disable dictionaries. The speak parameters can be updated using the **TtsSetParams** function. Note that a limited number of parameters can be updated while **TtsProcess(Ex)** is busy (rate and volume). When the input text contains markup controlling the speech generation, the parameters will be updated for the course of the current TtsProcess(Ex) execution, but will be reset to the values set via SetParam(s) and the dictionary API functions.
9. When using dictionaries, the Application calls the **TtsUnloadUsrDictEx** function for each dictionary that has been loaded by **TtsLoadUsrDictEx**. This unloads all dictionary instances in use by the application.
10. When maps of fetch properties are created for either SpeakData or DictionaryData, the Application calls the

TtsMapDestroy function for each map that has been created.
11. The Application calls the **TtsUninitialize** function to cleanup the TTS engine instance.

The RealSpeak v3.5 API did not support the xxxEx functions and the map functions. But we recommend using the new functions.

Demonstration applications

Standard Demo

This is a simple command-line demonstration application which runs on all platforms. It processes one text file in RealSpeak native input format (using native markup and native character set for the specified language). The output is one linear 16-bit PCM speech file (with 8kHz sample rate) named “standard.pcm”.
It should be run as follows:

standard <language> <voice> <engine directory> <text file>

Running the program without arguments, displays the usage and some examples.

Some examples:

standard “American English” Jill “%SSFTTTSSDK%\speech” “%SSFTTTSSDK%\api\demos\data\us_english.txt”

standard 0 0 “%SSFTTTSSDK%\speech” “%SSFTTTSSDK%\api\demos\data\us_english.txt”

The parameters (all required) are:

language	Language name or number (e.g. (e.g. 0 specifies “American English”)
voice	Voice name or number (e.g. 0 specifies the first female voice for the specified language)
engine directory	The speech subdirectory of the installation directory (which is specified by the SSFTTTSSDK environment variable). Specifying the installation directory also works.
text file	File name of input text in RealSpeak native input format

Chapter III

	(using native markup and native character set for the specified language)

The language and voice numbers are listed in "%SSFTTTSSDK%\api\inc\lh_ttsso.h".

The sources and makefiles are installed at "%SSFTTTSSDK%\api\demos\standard". For Windows, a Microsoft Developer Studio Project File (.dsp) is available, for UNIX a makefile to be used with the UINIX. make command is provided. The user has to comment out the appropriate compile line.

Some comments on the implementation

This source code of this program demonstrates the simplest possible call sequence to perform TTS using the RealSpeak v3 API functions: first create and initialize a TTS engine instance via **TtsInitialize**(), then process an input text via **TtsProcess**(), and finally destroy the TTS engine instance with **TtsUninitialize**().
This demo shows how the language and voice can already be specified when the TTS engine instance is created.
When operating RealSpeak in in-process mode, the application must specify the path of the engine library using the *szLibLocation* member of the TTSPARM structure passed into the **TtsInitialize** function. See the "API reference" chapter for a description the other members of the TTSPARM structure that are set in the demo.
This application provides text input to RealSpeak via the source call-back implemented by the function CbTtsSource. The output is received via the call-back function CbTtsDestination. Events or markers are received via the CbTtsEventNotify function which ignores all events. All RealSpeak call-backs are specified when TtsInitialize() is called.

Standardex Demo

This is a simple command-line demonstration application which runs on all platforms. It processes one text document which can be an SSML document or a text with RealSpeak native markup.
The text can be presented in whatever character set supported for the specified language. The input text can be specified via a local filename or via a URI that refers to a document on an HTTP server. When internet fetching is used, proxy and disk cache properties can be specified.
The output is one linear 16-bit PCM speech file (with 8kHz sample rate) named "standardex.pcm".
It should be run as follows:

standardex <language> <voice> <engine directory>
((<input: text file> <content-type>) or <URI>)
((<proxy server> <proxy port>) or 0) <CachePath>

Running the program without arguments, displays the usage.
Some examples:
(assuming the demo application is run from the %SSFTTTSSDK% directory)

- Input from local text file (possibly containing native markup):
  standardex "American English" Jennifer ./speech ./api/demos/data/us_english.txt text/plain;charset=iso-8859-1 0 ./speech/components/common/cache
- Input from local 4SML file:
  standardex German Steffi ./speech ./api/demos/data/ german_4sml.ssml application/ssml+xml 0 ./speech/components/common/cache
- Input from http:// URI, no use of internet fetch cache:
  standardex 0 0 ./speech http://arctic/realspeak/demo.ssml http://proxy01 80 0 0
- Input from http:// URI, enable the use of the internet fetch cache:
  standardex 0 0 ./speech http://nepal/realspeak/demo.txt http://proxy01 80 ./speech/components/common/cache 1000

The arguments are:

language	Language name or number (e.g. (e.g. 0 specifies "American English")
voice	Voice name or number (e.g. 0 specifies the first female voice for the specified language)
engine directory	The speech subdirectory of the install directory (which is usually specified by the SSFTTTSSDK environment variable)
input: text filename or URI	Input text provided via a filename or a URI. The markup format and the character set can be any of the supported ones (see next argument).
content-type	String that describes the type of content of the input. This argument is only required when the input is provided via a filename. For URI's that start with http:.// the argument must not be present. Then the content type is determined by the HTTP Server (only if http:// prefix) or by

	making use of the extension rules (e.g. .txt document is assumed to use the native markup and native character set, .ssml is an SSML document). The supported values for content-type are listed in the API Reference chapter, section "Defined Data Types", item "SpeakData". E.g. "text/plain;charset=windows-1252"
proxy server	Name of the proxy server to be used for internet fetching. Use value "0" if no proxy server has to be used.
proxy port	If <proxy server> is not "0", specifies the proxy port number
cache path	Directory name for the disk cache used when fetching URI's. Use value "0" if caching must be disabled.

The language and voice numbers are listed in "%SSFTTTSSDK%\api\inc\lh_ttsso.h".

The sources and makefiles are installed at "%SSFTTTSSDK%\api\demos\standardex". For Windows, a Microsoft Developer Studio Project File (.dsp) is available, for UNIX a makefile to be used with the UINIX. make command is provided. The user has to comment out the appropriate compile line.

Some comments on the implementation

This source code of this program demonstrates the use of the main RealSpeak v4 extended functions: first create and initialize a TTS engine instance via TtsInitializeEx(), then process an input text via TtsProcessEx(), and finally destroy the TTS engine instance with TtsUninitialize(). When calling TtsProcessEx(), a fetch property map is specified. The map is created using TtsMapCreate() and the download timeout property is set using TtsMapSetU32().

This application provides text input to RealSpeak via the SpeakData parameter of the ProcessEx() function: when the input is an HTTP URI, the URI is specified; else the content of the file is read into a buffer whose address is set in the data member and the contentType member refers to the content-type command-line argument.
The output is received via the call-back function CbTtsDestination. Events or markers are received via the CbTtsEventNotify function which ignores all events. All call-backs are specified when TtsInitialize() is called.

Chapter III

4SML Demo

This is a simple command-line demonstration application which runs on all platforms. It processes one 4SML text file (see “SSML support chapter). The output is one linear 16-bit PCM speech file (with 8kHz sample rate) named “4sml.pcm”. It is the counterpart of the “standard” demo for 4SML input. Note that 4SML processing can also be demonstrated with the standardex demo.
It should be run as follows:

4sml <language> <voice> <engine directory> <text file>

Running the program without arguments, displays the usage and some examples.

Some examples:

4sml “American English” Jill “%SSFTTTSSDK%\speech” “%SSFTTTSSDK%\api\demos\data\us_english_4sml.ssml”

standard 0 0 “%SSFTTTSSDK%\speech” “%SSFTTTSSDK%\api\demos\data\us_english_4sml.ssml”

The parameters (all required) are:

language	Language name or number (e.g. (e.g. 0 specifies “American English”)
voice	Voice name or number (e.g. 0 specifies the first female voice for the specified language)
engine directory	The speech subdirectory of the install directory (which is specified by the SSFTTTSSDK environment variable)
text file	Input text in 4SML format

The language and voice numbers are listed in “%SSFTTTSSDK%\api\inc\lh_ttsso.h”.

The sources and makefiles are installed at "%SSFTTTSSDK%\api\demos\4sml". For Windows, a Microsoft Developer Studio Project File (.dsp) is available, for UNIX a makefile to be used with the UINIX. make command is provided. The user has to comment out the appropriate compile line.

Some comments on the implementation

This source code of this program demonstrates the simplest possible call sequence to process an SSML or 4SML file using the RealSpeak

Chapter III

v3.5 API: first create and initialize a TTS engine instance via TtsInitialize() and change the markup type parameter to 4SML using TtsSetParam(), then process a 4SML text via TtsProcess(), and finally destroy the TTS engine instance with TtsUninitialize().
This application provides text input to RealSpeak via the source call-back implemented by the function CbTtsSource. The output is received via the call-back function CbTtsDestination. Events or markers are received via the CbTtsEventNotify function which ignores all events. All RealSpeak call-backs are specified when TtsInitialize() is called.

Client/server use of RealSpeak

Intro

RealSpeak Telecom supports Client/Server functionality. This configuration has already been explained in the "System Overview" section of the Introduction chapter and figures I-3 and I-4 illustrated this configuration.
The application designer links the TTS API Library to the application in just the same way as when using RealSpeak in-process, but the use of the API is slightly different as explained a bit further. The TTS server standalone program is run on one or more server machines on the network. Each server can have one or more RealSpeak voices installed.
The RealSpeak client instances communicate with the TTS server through a standard TCP/IP socket connection.

Running the TTS Server

Intro

The TTS Server executable is called ttsserver.exe on Windows, and ttsserver on all other operating systems. The RealSpeak Telecom installer installs it in the top-level of the RealSpeak Telecom installation directory, normally specified by the environment variable SSFTTTSSDK.

Configuring the server

The TTS Server is configured using an XML configuration file, by default $SSFTTTSSDK/config/ttsserver.xml (Unix) or %SSFTTTSSDK%\config\ttsserver.xml (Windows), which specifies a large number of configuration parameters. Some examples of parameters are: the TCP port number of the RealSpeak service, the location of the license server, the default speech volume and internet fetch properties.
To run the server with the ScanSoft provided default configuration,

change to the directory where the server is installed and run the TTS server executable with no arguments, such as:

(Windows) ttsserver.exe
(Unix) ttsserver

To run the server with a modified configuration, there are two options. The first is to modify the default configuration file ttsserver.xml, then start the server as shown above. However, if you do so and later install a RealSpeak Telecom service pack or upgrade, your ttsserver.xml will get overwritten. The second option is to create a second XML configuration file and use it to augment the default configuration file. To do so, make a site-specific copy of ttsserver.xml, such as ttsserver_site.xml. Then open the copy in a text or XML editor and remove all the parameters except the specific ones you wish to customize. Then customize those parameters and save it. Finally, start the TTS Server specifying both configuration files, where parameters found in the file specified by the second -c argument (the site-specific settings) override the ScanSoft provided defaults. The TTS Server allows you to specify as many -c options as you like, permitting multi-level configuration file inheritance.

(Windows) ttsserver.exe -c %SSFTTTSSDK%\config\ttsserver.xml -c %SSFTTTSSDK%\config\ttsserver_site.xml

(Unix) ttsserver -c $SSFTTTSSDK/config/ttsserver.xml -c $SSFTTTSSDK/config/ttsserver_site.xml

For details on the TTS Server configuration parameters, see the "Configuration Files" section in the "User Configuration" chapter.

Specifying the installation directory

In Client/Server mode, the TTS Server looks for the TTS engine library in the directory specified by the -d option when the server program was started. If the -d option was not specified, it looks for the engine in the default directory. For the TTS Server, it's also possible to set the SSFTTTSSDK environment variable in the configuration file. Please note that the environment variable, SSFTTTSSDK is automatically set by the windows installer. This does not apply to the UNIX installers.

API Call sequence

An engine instance can be opened in Client/Server mode using the **TtsCreateEngine** function, which sets up a connection with a TTS Server running on a network host., followed by a **TtsInitializeEx** call that specifies the server argument returned by the TtsCreateEngine call. After this point, the call sequence is the same as

Chapter III

for in-process use of RealSpeak. The only exception is that after closing a client instance with **TtsUnitialize, TtsRemoveEngine** should be invoked to close the connection with the server. Note that this procedure is in prinicipal no longer required with RealSpeak v4.0, since TtsCreateEngine and TtsRemoveEngine have become dummy functions.

Demonstration applications

Twonode Demo

This is a simple command-line application which runs on all RealSpeak platforms and demonstrates the client/server operation of RealSpeak. It processes one text file in RealSpeak native input format. The output is one linear 16-bit PCM speech file with an 8kHz sample rate named “twonode.pcm”.
Before running the program, one must start the TTS Server application on the server machine and configure it to provide the TTS service via the TCP port number 6666. This can for instance be performed by running the below command from the RealSpeak installation directory.

ttsserver -p6666 -d.

Then on the client machine run the twonode demo program as follows:

twonode <language> <voice> <server> <text file name>

Running the program without arguments, displays the usage.

Some examples:

twonode “American English” Jill pegasus
“%SSFTTTSSDK%\api\demos\data\us_english.txt”

twonode 0 0 10.0.1.10 “%SSFTTTSSDK%\speech”
“%SSFTTTSSDK%\api\demos\data\us_english.txt”

Chapter III

The parameters (all required) are:

language	Language name or number (e.g. (e.g. 0 specifies “American English”)
voice	Voice name or number (e.g. 0 specifies the first female voice for the specified language)
server	Name or IP address of TTS server; the port number is fixed at 6666.
text file name	File name of input text in RealSpeak native input format (using native markup and native character set for the specified language)

The language and voice numbers are listed in “%SSFTTTSSDK%\api\inc\lh_ttsso.h”.

The sources and makefiles are installed at "%SSFTTTSSDK%\api\demos\twonode". For Windows, a Microsoft Developer Studio Project File (.dsp) is available, for UNIX a makefile to be used with the UINIX. make command is provided. The user has to comment out the appropriate compile line.

Some comments on the implementation

The source code of this program demonstrates the use of a TTS server. This program first calls the TtsCreateEngine API function as explained in the “API Call sequence” section above.
Note that the Parm.szLibLocation field when calling TtsInitialize() is NULL. This is allowed because TTS is used in client/server mode; in this case the RealSpeak client will use the SSFTTTSSDK environment variable to determinate the engine directory.

Dict_n_rules Demo

This is a command-line demonstration application which runs on all platforms. It demonstrates the deployment of user dictionaries and rulesets. The demo can be run as a TTS client when the ‘-s’; option is set, else it runs TTS in-process. It generates linear 16-bit PCM files.

It processes a list of text documents which can be SSML documents or texts with RealSpeak native markup.
The input texts can be specified via a local filename or via a URI that refers to a document on an HTTP server. A base URI or local file can be specified.
The output are linear 16bit PCM speech files (with 8kHz sample rate), one for each input document, named “<root input name without extension>.pcm” and written to the specified output directory.
It should be run as follows:

dict_n_rules -l <language> [-v <voice>] -@ <input text list file> [-d <install directory>] [-o <output directory>] [-u <user dictionary list file>] [-r <ruleset list file>] [-b <base URL>] [-s <server name>]

Running the program without arguments, displays the usage.
Some examples:
(assuming the demo application is run from the %SSFTTTSSDK% directory)

dict_n_rules -l "American English" -v Jill -@ input.lst -u dict.lst -r ruleset.lst
(in-process use of TTS)
dict_n_rules -l "American English" -v Jill -@ input.lst -u dict.lst -r ruleset.lst -s Pegasus
(Client/Server use of TTS)

The arguments are:

-l <language>	[required] Language name or number (e.g. (e.g. 0 specifies "American English")
-v <voice>	[optional] Voice name or number (e.g. 0 specifies the first female voice for the specified language) Default: 0, the first female voice
-d <install directory>	[optional] The installation directory (which is usually specified by the SSFTTTSSDK environment variable). Default: the value of the environment variable SSFTTTSSDK
-@ <input text list file>	[required] File listing one or more input documents; the format of each line: URI or local file name with optional MIME content-type (default: left unspecified). If a file name contains spaces it must be enclosed by double quotes. The content-type parameter allows the specification of the markup format and the character set, see SpeakData structure for a description of the supported values. Note that for URI's that start with http:.//, RealSpeak can usually determine the content-type automatically.

Chapter III

-o <output directory>	[optional] Default: the current working directory
-u <user dictionary list file>	[optional] File listing one or more user dictionaries; format of each line: URI or local file with optional MIME-type (default: left unspecified) Default: no dictionaries
-r <ruleset list file>	[optional]File listing one or more user rulesets; format of each line: URI or local file with optional MIME-type (default: left unspecified) Default: no rulesets
-b <base URL>	Base URI or local file, the path of this URI or file is used when loading user dictionaries and rulesets that are specified with a relative path name. Default: undefined
-s <server name>	Run in client/server mode using the specified host (name or IP address) as server; the port number is fixed at 6666. Default: undefined, run TTS in-process

The language and voice numbers are listed in "%SSFTTTSSDK%\api\inc\lh_ttsso.h".

The sources and makefiles are installed at "%SSFTTTSSDK%\api\demos\dict_n_rules". For Windows, a Microsoft Developer Studio Project File (.dsp) is available, for UNIX a makefile to be used with the UINIX. make command is provided. The user has to comment out the appropriate compile line.

Some comments on the implementation

This source code of this program demonstrates the use of user dictionaries and rulesets.

The file "dict_n_rules.c" implements the main() function which contains all RealSpeak API calls.
The file "listfile.c" implements the class of list files which can be used to parse a list file and iterate through it.
If the client/server mode is enabled, the TtsCreateEngine() function is called first. From that moment, the call sequence is the same as for in-process mode, apart from the calling of TtsRemoveEngine() at the end of the program.
A TTS engine instance is created via TtsInitializeEx().

Chapter III

Then a fetch property map is constructed with TtsMapCreate() and, if specified on the command line, the URL base and the download timeout property are set via the appropriate TtsMapSetXxx() function.
Then each user dictionary specified in the User dictionary list file is loaded and implicitely enabled with the TtsLoadUsrDictEx() function. If appropriate, the fetch properties are applied during the loading.

Each user ruleset specified in the Ruleset list file is loaded and implicitely enabled with the TtsSetParams() function. If appropriate, the fetch properties are applied.

Then, each input text document (specified via a file or URI) listed in the Input text list file, is processed via TtsProcessEx().
This application specifies the input to RealSpeak via the SpeakData parameter of the ProcessEx() function.: the uri member refers to the filename or URI. contentType member refers to the content-type specified in the Input list file. It is set to NULL if the content-type has not been specified., and then the engine will determine the type automatically.

Subsequently, all user dictionaries are disabled with one call to the TtsDisableUsrDictsEx() function. The dictionary engine instances are unloaded one by one using the TtsUnloadUsrDictEx() function. The user rulesets are unloaded one by one using the TtsSetParams() function.
Then the fetch property map is destroyed using TtsMapDestroy() function.
Finally the TTS engine instance is destroyed with TtsUninitialize(), followed by a TtsRemoveEngine() call if a TTS server was used.

The output is received via the call-back function CbTtsDestination. Events or markers are received via the CbTtsEventNotify function which ignores all events. All call-backs are specified when TtsInitialize() is called.

RealSpeak Parameters

Introduction

In a client/server environment, the default values for numerous parameters can be set in one or more RealSpeak Server configuration files.
When a TTS engine instance is created using the TtsInitialize(Ex) API function, the majority of these parameters and some extra parameters can be set to new values, overriding the default values of the configuration files.

Chapter III

After creation, most "speak" parameters can be updated using the TtsSetParam(s) API functions.
Some parameters can be set for a particular Speak request when calling the TtsProcessEx() function.
Most speak parameters can also be changed by marking up the input text. If so, those values override both the RealSpeak Server defaults and the value set via the API. But they are only active for that (and only that) speak request.

Use of Configuration Files

The TTS Server is configured using an XML configuration file, by default "config/ttsserver.xml" within the RealSpeak Telecom installation directory.
Note that when operating RealSpeak in in-process mode, the server configuration file is not used at all (except when using the SAPI5 interface). The use of server configuration files is explained in more detail in the "Running the TTS Server" subsection of the "Client/Server Use of RealSpeak" section further in this chapter.

Setting of Parameters via the API

Non-speak parameters

The parameters that are not directly related to the text-to-speech conversion process, are usually shared over all engine instances. They can be set via the TtsInitialize(Ex) API function. Note this function initializes a certain instance, but only the first call to TtsInitialize(Ex) can set the concerning parameters. Examples of this class of parameters are the "RealSpeak Installation directory" and the "cache directory" for internet fetches.

Speak Parameters

Parameters tuning the Text-to-speech conversion that are set via the API apply to a certain **instance**. A number of parameters can already be specified when the engine instance is **initialized** via the TtsInitialize(Ex) API function, by setting the appropriate fields in a "TTSPARAM" structure. This structure specifies for instance the initial language and voice and , the volume. As mentioned above this structure also specifies parameters that are shared over all instances. The structure type will be discussed in greater detail in the "TTSPARM" topic in the "RealSpeak API" chapter.

Examples:

```
TtsParm.nLanguage = TTS_VOICE_USE_STRING;
TtsParm.szLanguageString = "French";
TtsParm.nVoice = TTS_VOICE_USE_STRING;
TtsParm.szVoiceString = "Sophie";
```

Chapter III

Most speak parameters can be set using the TtsSetParam(s) functions. In most cases parameters cannot be updated when the instance is busy executing the TtsProcess(Ex) function (which performs the text-to-speech conversion process) in another thread. Only the volume and the rate can be updated while speaking

Text Markup

Finally, a lot of the parameters can also be updated by inserting markup in the input text. The RealSpeak API supports two markup languages: the native one (the default) and 4SML (SSML with some proprietary extensions). The same markup languages are supported when the Speechify API is used.
SAPI5 is supported when the SAPI interface is used. Please refer to the "SAPI5 Compliance" and "SSML Support" chapter for details regarding the SAPI5 or SSML markup languages. The native markup language is described in each of the language specific user guides.

Overview of RealSpeak parameters

The table below gives an overview of all the RealSpeak parameters and how they can be set. Setting parameters via environment variables or the registry is however not described; please refer to the "Installation Guide" and the "SAPI5 compliance" chapter for this.

As an example of how to interprete the table, here is a description of the setting of the "voice" speak parameter.
RealSpeak allows the selection of the voice in three different ways; it however does not support the setting of a "default voice" via the configuration files or as a parameter of the TtsProcess(Ex) function.
The voice can be set:

- Via the TtsInitialize(Ex) API function, by setting the appropriate values in the TTSPARAM structure. For the details see the description of the TTSPARAM structure type in the "RealSpeak API" chapter; the involved structure fields are 'nVoice' and 'szVoiceString'
  Example:
  ```
  Parm.nVoice = TTS_VOICE_USE_STRING;
  Parm.szVoiceString = "Jill";
  ```
- Using the TtsSetParam(s) function, providing the instance is not busy executing the TtsProcess(Ex) function in another thread.
- Using markup: the SSML <voice> element and voice attribute, the SAPI5 voice element or the native <esc>\voice\ tag.

Most parameters that can be changed via the TtsSetParam (s) functions can only be updated while the TTS instance is in an idle

state, meaning that there is no TtsProcess(Ex) function being executed for that instance in another thread. The only two exceptions are the rate and volume parameters

Parameter	Configuration Files	Initialize function	SetParam(s) function	ProcessEx function (SpeakData parameter)	Text Markup
Environment Settings					
Installation directory	Y (<SSFTTTSSDK>)	Y (shared over all instances; TtsParm.szLibLocation)	N	N	N
Temp. files directory	Y (<TMPDIR>)	N	N	N	N
User ID	Y (<USER>)	N	N	N	N
Network parameters					
See "User Configuration" chapter for the full list. E.g. TCP/IP port number	Y (for all)	N (for all)	N (for all)	N (for all)	N (for all)
TTS servers	The SAPI5 client configuration file must list the available TTS servers under <tts_servers>	Y (the client specifies the hostname of the server, the port number or the service name in the TTS_SERVER structure.)	N	N	N

Chapter III

Parameter	Configuration Files	Initialize function	SetParam(s) function	ProcessEx function (SpeakData parameter)	Text Markup
Licensing Parameters					
Licensing mode, default or explicit	Y (<license_mode>)	N	Y; idle state (TTS_LICENSE_MODE_PARAM)	N	N
License servers	Y (<license_servers>	N	N (use the SSFT_TTS_LICENSE_SERVERS environment variable when using RealSpeak in-process)	N	N
Miscellaneous Server Parameters					
See "User Configuration" chapter for the full list. E.g. default mode for running the server (background versus interactive)	Y (for all)	N (for all)	N (for all)	N (for all)	N (for all)
Diagnostic and Error Logging Parameters					
"User Configuration" chapter for the full list. E.g. log level, maximum size of log file	Y (for all)	N (for all)	N (for all)	N (for all)	N (for all)

Chapter III

Parameter	Configuration Files	Initialize function	SetParam(s) function	ProcessEx function (SpeakData parameter)	Text Markup
Internet Fetch parameters					
Global cache parameters (E.g. cache directory, cache max. total size cache). See TTSPARAM and "User Configuration" chapter for more details	Y (E.g. <cache_directory>, <cache_total_size>)	Y (E.g. szCachePath, CacheTotalSizeMb members of TtsParm); shared over all instances	N	N	
Proxy parameters (proxy server and port number)	Y (<inet_proxy_server> and <inet_proxy_server_port>)	Y (szProxyServer and nProxyPortNumber member of TtsParm; shared over all instances)	N	N	N
Inet extension rules	Y (<inet_extension_rules>)	N	N	N	N
User agent name in HTTP headers	Y (<inet_user_agent>)	N	N	N	N
Whether to accept HTTP cookies	Y (<inet_accept_cookies>)	N	N	User specific cookie jar can be provided via fetchCookieJar parameter; when NULL cookies are refused	N
URL base	N	N	N	Y ("inet.urlBase" in fetchProperties)	Y for 4SML

Chapter III

Parameter	Configuration Files	Initialize function	SetParam(s) function	ProcessEx function (SpeakData parameter)	Text Markup
Web server fetch timeout	N	N	N	Y (fetchProperties)	Y for 4SML (fetchtimeout for <audio>)
Use of cache entry for a specific internet fetch	N	N	N	Y(fetchProperties "inet.maxage" and "inet.maxstale")	Y for 4SML (maxage and maxstale for <audio>)
Input text parameters					
Markup type (native or 4SML for RealSpeak API, SAPI5 is only possible if SAPI interface is used)	N	N	Y; idle state (TTS_MARKUP_TYPE_PARAM)	Y (contentTyp)	N
Character encoding	N	N	N	Y (contentTyp)	Y for 4SML and SAPI5 (<?xml> header)
Type of document (e.g. "text" or "email")	N	N	Y; idle state (TTS_DOCUMENT_TYPE_PARAM)	N	Y for 4SML (ssft-dtype attribute) and native markup (<esc>%)
Audio Parameters					
Sample frequency	N	Y (TtsParm.nFrequency)	N	N	N
Sample format (e.g. mu-law)	N	Y (TtsParm.nOutputType)	Y; idle state (TTS_OUTPUT_TYPE_PARAM)	N	N

Parameter	Configuration Files	Initialize function	SetParam(s) function	ProcessEx function (SpeakData parameter)	Text Markup
Event parameters					
call-back functions	N	Y (TtsParm.cbFuncs)	N	N	N
Marker mode	N	N	Y; idle state (TTS_MARKUP_TYPE_PARAM)	N	N
Speak Parameters					
Language (language and country or dialect)	N	Y (TtsParm.szLanguageString or TtsParm.nLanguage)	Y; idle state (TTS_LANGUAGE_PARAM)	N	Y for 4SML (xml:lang attribute) and SAPI5 (<Lang>)
Voice (name, gender, number, etc)	N	Y (TtsParm.szVoiceString or TtsParm.nVoice)	Y; idle state (TTS_VOICE_PARAM)	N	Y (all)
Rate	Y (<default_rate>)	N	Y; idle and busy state (TTS_RATE_LARGESCALE_PARAM or TTS_RATE_PARAM)	N	Y (all)
Volume	Y (<default_volume>)	N	Y; idle and busy state (TTS_VOLUME_LARGESCALE_PARAM or TTS_VOLUME_PARAM)	N	Y (all)

Chapter III

Parameter	Configuration Files	Initialize function	SetParam(s) function	ProcessEx function (SpeakData parameter)	Text Markup
Pitch (not supported by the engine)	N	N	Y (TTS_PITCH_PARAM) but not supported by engine	N	Y but not supported by engine
Read mode (e.g.sentence-by-sentence, word-by-word)	N	N	N	N	Y for native markup
Spell mode on/off	N	N	N	N	Y (all)
End-of-message pause length	N	N	N	N	Y for native markup

Use of RealSpeak in telephone or dialogue applications

Multiple engine instances

Telephony applications are designed to service many customers at the same time. The concept of "voice port" is often used in this domain. Each voice port can service one customer at a time.
Typically, a telephony application will direct all TTS requests for one telephone call or dialogue session to the same TTS engine instance. The term "call" will be used to refer to a telephone call by a customer or any form of dialogue session.
The instance can in principle be created and initialized at the start of the call and destroyed when the call is terminated. But it is usually more efficient to keep the engine instance alive and reuse it for another call; this means one TTS engine instance is assigned to one voice port for a lifetime that is usually much longer than one call.

Note that reusing an engine instance for a new call usually requires the application to restore the engine instance to a well-known state before reusing it for another call. This is needed when the settings of an instance are changed in the course of one call (e.g. adjust the volume, switching the voice or language, loading of user dictionaries). These changes are undone in a similar way as they were performed: using the TtsSetParams() function and the API functions for the enabling/disabling of user dictionaries.

Chapter III

Since the TtsProcess() function is a synchronous or blocking function, the concurrent handling of multiple voice ports, requires the use of process threads. Usually one thread is created for each voice port. The audio is streamed back to the telephone application via the Destination call-back. The first argument of which, the application data pointer, can be used to direct the audio to the appropriate voice port.

Real-time responsiveness and audio streaming

To support real-time audio streaming, the engine should return audio chunks at a rate that is faster or equal to the play-back rate.
When servicing multiple voice ports each TTS instance should be run in a separate thread to enable real-time audio streaming for all ports. On the server, threads are automatically created for each server engine instance. By using threads, the switching between the instances is handled by the operating system.

The RealSpeak engine attempts to minimize the latency for each TTS request by sending audio back as soon as the buffer provided by the application can be filled completely.
The engine instance achieves this by narrowing the window moving over the data as much as possible without degrading the speech quality. One of the first processing steps is to split off a next sentence. Then the linguistic processing is performed on that sentence. The unit selection normally operates on one sentence, but for long sentences, it limits its window to one phrase. The final subprocess, the synthesizer, processes one speech unit at a time and narrows its scope to a small chunk of speech samples when nearing the output step.
As soon as that chunk of speech is sent to the application via the Destination call-back, it can be played back. The Destination call-back should return as soon as possible, to allow the instance to process the rest of the speech unit, sentence or text and fill a next buffer with audio before the audio of the previous buffer has played out. Note that the size of the output chunk is determined by the application, for reasons of efficiency it should not be too small, but to minimize the latency it shouldn't be too big. A good compromise is 4Kbytes.
The above explains that the latency for the first audio chunk of a sentence is usually longer than for the following chunks, which is convenient since the effect of an underrun at the start of a sentence is less critical: it results in a longer pause in-between two sentences.
The latency also depends on the type of input text. The application designer should provision RealSpeak with an adequate safety margin for the possible variance in the latency. For instance, if most TTS requests consist of a text with normal sentences but a few may have extremely long sentences (e.g. poorly punctuated e-mails) then allowance should be made for situations where the TTS engine

instance will have to deal with long sections of text with no punctuation. Such an occurrence may result in an extended inter-sentence latency, normally audible as a longer pause in-between two sentences. To reduce the risk for extended inter-sentence latencies, the engine will split up very long sentences at an appropriate location (e.g. a phrase boundary). But is exceptional to have a natural sentence that long (the length depends on the language and the sentence's content and but it's usually around 750 characters).
Note that when servicing multiple voice ports and assuming each TTS instance is run in a separate thread, instances are not influenced by the badly punctuated input of another instance.
If an audio chunk is not returned before the previous chunk has been played out, a gap in the speech output will be heard. Such an intra-sentence gap can have a stronger audible effect: it can occur for instance within a word. But such an "underrun" is less likely, and will only start to appear when operating RealSpeak near its limits. The effects can be masked by maintaining a queue of audio chunks which allows accumulating audio faster than real-time to compensate for the rare occasion of a non-realtime response. Usually the audio output device will support the queuing or buffering of audio chunks before they are effectively played out. In any case, when the Destination call-back returns with an audio chunk, it's safer to return a buffer for the next chunk ASAP instead of waiting till the play-back of the previous chunk has finished. Of course, when the instance runs faster than real-time, the insertion of some waiting can be appropriate when the size of the queue grows too much (a fixed maximum on the number of buffers would then result in an overrun). Note that because of the 'throttling' mechanism implemented by RealSpeak, the audio chunk delivery rate is limited to two times real-time, thus reducing the risk for overruns.

RealSpeak Telecom
Software Development Kit

Chapter IV

RealSpeak API

Programmer's Guide

RealSpeak API

New and Changed in RealSpeak 4.0 API

As a result of the improved functionality, the interface of the TTS system has been changed. To ensure backwards compatibility, new functions and data types are added to the interface to provide the new functionality; both behavior and interface remain the same for the existing functions and data types. Some new functions are an extension of existing functions and it is recommended to use those instead of the existing ones. Above that, the new functionality includes a run-time based licensing system. This paragraph contains an overview of all changes.

The following functions are new for this release:

TtsDisableUsrDictsEx
TtsMapCreate
TtsMapDestroy
TtsMapSetChar
TtsMapSetU32
TtsMapSetBool
TtsMapGetChar
TtsMapFreeChar
TtsMapGetU32
TtsMapGetBool
TtsSetParams
TtsGetParams
TtsResourceAllocate
TtsResourceFree

The following functions are an extension of existing functions. They end with an 'Ex' to make a clear separation between the old and new functions. The corresponding old functions can be found between brackets:

TtsInitializeEx (TtsInitialize)
TtsProcessEx (TtsProcess)
TtsLoadUsrDictEx (TtsLoadUsrDict)
TtsUnloadUsrDictEx (TtsUnloadUsrDict)
TtsEnableUsrDictEx (TtsEnableUsrDict)
TtsDisableUsrDictEx (TtsDisableUsrDict)

The following data types are new for this release:

HTTSDCTEG
HTTSMAP
HTTSVECTOR
TTS_PARAM_T
TTS_Marker
TTS_Event
TTS_BookMark
TTS_SentenceMark

TTS_WordMark
TTS_PhonemeMark
TTS_ParagraphMark
SpeakData
DictionaryData

HTTSDICT and **HTTSDCTEG** are equivalent; they are both handles to a dictionary instance. By design, HTTSDCTEG must be used in combination with the new functions while HTTSDICT must be used in combination with the already existing functions.

The **SpeakData** structure is used in combination with the new **TtsProcessEx** function.
It is used to describe the actual TTS input data: its location, fetch properties and type (used markup language and character set). This new approach enables references to URI addresses and supports the specification of the document type: "standard" (use of native markup) versus 4SML (or SSML). Fetch properties can be especially handy in case of remote access to the input data (e.g. for specifying a fetch timeout). The input text can also be provided via a memory buffer. When SpeakData is used, it replaces the TtsSourceCb callback function; the source callback can still be used with the new function, though
The **DictionaryData** structure is used in a similar way as the SpeakData structure; it describes a dictionary: its location, fetch properties, type and priority.

The **TTSPARM** data type has been extended and contains now more members. Basically there are two kinds of new members: parameters that describe the new proxy server functionality and parameters that describe the new cache functionality. The existing functions will only use the old members.

The **TTS_PARAM_T** data type has to be used together with TtsGet/SetParams and makes it possible to respectively query or modify multiple parameters from an engine instance with one function call.

The TTS_Marker, TTS_Event, TTS_BookMark, TTS_SentenceMark, TTS_WordMark, TTS_PhonemeMark and TTS_ParagraphMark are used to support **markers** or events. Markers can be received via the TtsEventCb callback function and provide the application with extra info about the message that is being processed. The callback was already defined in the previous version, but it was never called.

Run-time **licensing** has been enforced to the product. The new API function **TtsResourceAllocate** allows control on the use of licenses. By default implicit licensing mode is used which requires no extra function calls.

The licensing is based on the number of active TTS engine instances. There are two licensing modes, an explicit and an implicit one (the default). More info about licensing can be found in the "Installation Guide" chapter.

Note: don't use the old functions and the new functions in conjunction!

Defined Data Types

This section describes the defined data types that are required to interact with the API.

HTTSDICT

HTTSDICT is the type representing the handle to a dictionary instance. It is returned from a successful call to the TtsLoadUsrDict function. This type is in the header file lh_ttsso.h.

This type is used in combination with the obsolete TtsLoadUsrDict/TtsUnloadUsrDict functions and should not be used in new implementations anymore; see HTTSDCTEG.

HTTSDCTEG

HTTSDCTEG is the type representing the handle to a dictionary instance. It is returned from a successful call to the TtsLoadUsrDictEx function. This type is in the header file lh_ttsso.h.

HTTSDICT and HTTSDCTEG are the same. By design, HTTSDCTEG must be used in combination with the new functions.

HTTSINSTANCE

HTTSINSTANCE is the type representing the handle to an open TTS engine instance. It is returned from a successful call to the TtsInitialize or TtsInitializeEx function. This type is in the header file lh_ttsso.h.

HTTSMAP

HTTSMAP is the type representing the handle to an open TTS map. A TTS map contains the fetch properties for speech data or a dictionary instance. It is returned from a successful call to the TtsMapCreate function. This type is in the header file lh_ttsso.h.

HTTSVECTOR

HTTSVECTOR is the type representing the handle to an open Vector. It is a member of the SpeakData and DictionaryData structures and is updated when the API is called.

TTSRETVAL

TTSRETVAL is the type representing a TTS error. This type is in the header lh_ttsso.h.

LH_SERVER_INFO

This structure is used to set the server network information. This structure is in the header lh_ttsso.h.

```
typedef struct
{
LH_CHAR         IP_Address[80];
LH_CHAR         service[80];
LH_S32          port_number;
} LH_SERVER_INFO;
```

Structure members

IP_Address	IP address of the server
Service	Service name
port_number	Port number

LH_SDK_SERVER

This structure holds the LH_SERVER_INFO structure with the server's network information in it and the handle to the server. This structure is in the header lh_ttsso.h.

```
typedef struct
{
LH_SERVER_INFO  server;
LH_S32          server_handle;
}LH_SDK_SERVER;
```

Structure members

server	Server information structure
server_handle	Handle to server

TTSCallBacks

TTSCallBacks is used to pass the addresses of callback functions in TtsInitialize() or TtsInitializeEx(). The function definitions for the callbacks are described in the User Callbacks section.

```
typedef struct
{
int                     numCallbacks;
TTSSOURCECB             TtsSourceCb;
TTSDESTCB               TtsDestCb;
TTSEVENTCB              TtsEventCb;
} TTSCallBacks;
```

Structure members

numCallbacks	Number of callbacks; is not used.
TtsSourceCb	Callback for source text
TtsDestCb	Callback for output audio
TtsEventCb	Callback for events

TTSPARM

A TTSPARM typed structure is used to specify the (initial) parameters for a given engine instance when calling TtsInitialize or TtsInitializeEx.. This data structure is defined in the header file "lh_ttsso.h". See Appendix A for a summary of the acceptable values for each member.

```
typedef struct
{
U16                     nLanguage;
CHAR*                   szLanguageString
U16                     nOutputType;
U16                     nFrequency;
U16                     nVoice;
CHAR*                   szVoiceString
CHAR*                   szLibLocation
U16                     nOutputDataType;
U16                     nInputDataType;
TTSCallBacks            cbFuncs;
CHAR*                   szProxyServer;
U32                     nProxyPortNumber;
CHAR*                   szCachePath;
U32                     nCacheTotalSizeMb;
U32                     nCacheEntryMaxSizeMb;
U32                     nCacheEntryExpTimeSec;
U32                     nCacheLowWaterMB;
int                     bCache;
} TTSPARM;
```

Structure members

nLanguage	Language to use, a value of TTS_LANG_USE_STRING instructs the API to use szLanguageString instead of nLanguage
szLanguageString	String specifying language to use (valid when nLanguage = TTS_LANG_USE_STRING). e.g. "American English"
nOutputType	Output type to use. Possible values: TTS_LINEAR_16BIT (16-bit linear), TTS_MULAW_8BIT (mu-law) or TTS_ALAW_8BIT (A-law)
nFrequency	Frequency to use. Possible values: TTS_FREQ_8KHZ (8kHz), TTS_FREQ_11KHZ (11kHz) or TTS_FREQ_22KHZ (22kHz)
nVoice	Integer specifying the voice to use, setting the value equal to TTS_VOICE_USE_STRING instructs the API to use the *szVoiceString* member instead of *nVoice*
szVoiceString	String specifying voice to use (only valid when *nVoice* = TTS_VOICE_USE_STRING). e.g. "Jennifer"
szLibLocation	Location of the engine libraries and databases; it should specify the path to the speech subdirectory of the RealSpeak installation directory (the latter one is specified by $SSFTTTSSDK or %SSFTTTSSDK%) Specifying the installation directory itself also works. Only in client/server mode, this member can be set to NULL, and then the client will use the SSFTTTSSDK environment variable. The specified value is never used by the server engine instance. The server uses the SSFTTTSSDK parameter specified in the server configuration file.
nOutputDataType	Not in use.
nInputDataType	Not in use.
cbFuncs	Pointers to application callback

functions, see TTSCallBacks
data type topic for more details

The following structure members are new for this release:

szProxyServer	String specifying proxy server to use. This value can be NULL when no proxy server is available.
nProxyPortNumber	The port number of the proxy server to use. Has only to be specified when szProxyServer is not equal to NULL.
szCachePath	String specifying the path of the cache. Subdirectory \cache is created automatically and should not be included in the cache path. Only required when bCache = true.
nCacheTotalSizeMb	The total size of the cache in Mb. Only required when bCache = true.
nCacheEntryMaxSizeMb	The maximum size a cache entry can have. Only required when bCache = true.
nCacheEntryExpTimeSec	Maximum amount of time any individual cache entry will remain in the cache, in seconds. Only required when bCache = true.
nCacheLowWaterMB	The minimum size the cache can have; everything above this threshold will be cleaned up after each action. This value can be 0.
bCache	Enable or disable the cache. When enabling the cache, the following structure members must have a value different from 0: nCacheTotalSizeMb, nCacheEntryMaxSizeMb, nCacheEntryExpTimeSec, nCacheLowWaterMB, szCachePath.

TTS_PARAM

TTS_PARAM is used to indicate the parameter type when calling TtsSetParam(s) and TtsGetParam(s). The members TTS_LANGUAGE_PARAM and TTS_VOICE_PARAM are introduced in version 4.0.

```
typedef enum
{
TTS_LANGUAGE_PARAM,
TTS_VOICE_PARAM,
TTS_VOLUME_PARAM,
```

```
TTS_RATE_PARAM,
TTS_PITCH_PARAM,                    /* NOT SUPPORTED
                                    IN REALSPEAK */
TTS_DOCUMENT_TYPE_PARAM,
TTS_VOLUME_LARGESCALE_PA
RAM,
TTS_RATE_LARGESCALE_PARA
M,
TTS_MARKUP_TYPE_PARAM,
TTS_OUTPUT_TYPE_PARAM,
TTS_MARKER_MODE_PARAM,
TTS_BLADE_ENABLE_PARAM,
TTS_BLADE_DISABLE,
TTS_LICENSE_MODE_PARAM,
TTS_RULESET_LOAD_PARAM
TTS_RULESET_UNLOAD_PARAM
TTS_TOTAL_PARAMS
} TTS_PARAM;
```

TTS_PARAM_VALUE_T

TTS_PARAM_VALUE_T union type is used as a storage place for the value of a parameter that is specified via a TTS_PARAM typed structure.

```
typedef union TTS_PARAM_VAL_U {
{
U32                        nNo;
TTS_PARAM_VAL_ARRAY_T      array;
VOID *                     pObj;
HTTSMAP                    hMap
} TTS_PARAM_VALUE_T;
```

With

```
typedef struct TTS_PARAM_S {
{
U16                        nValue;
void*                      pValue
}
TTS_PARAM_VAL_ARRAY_T;
```

TTS_PARAM_T

TTS_PARAM_T describes one parameter name, parameter value couple. This data structure is defined in the header file lh_ttsso.h. See the description of the TTS_PARAM type for a list of possible values for the nParam field.

```
typedef struct TTS_PARAM_S {
{
TTS_PARAM                  nParam;
```

```
TTS_PARAM_VALUE_T                    paramValue;
} TTS_PARAM_T;
```

TTS_FETCHINFO_T

TTS_FETCHINFO_T is used to provide all the information to load (fetch) or unload a ruleset.

```
typedef struct
{
const char*                    szUri;
const char*                    szContentType;
HTTSMAP                        hFetchProperties;
} TTS_FETCHINFO_T;
```

Structure members

szUri	String (zero-terminated) specifying the location of the ruleset document. This can be an http address (http://) or a file name (regular or with file://).
szContentType	String specifying the content-type of the ruleset. It's optional: specify NULL if not used. By default the content-type is assumed to be "application/x-realspeak-rettt+text" (see definition of RULESET_MIME_RETTT_TEXT in "lh_inettypes.h". szContentType can have the following constants as value: Value — File type RULESET_MIME_RETTT_TEXT — Textual RETTT ruleset Note that the character set does not need to be specified. Warning: rulesets must be encoded in the native character set for the TTS language specified in the header section of the ruleset. See the table in the “RealSpeak Languages” appendix for an overview of the native character set for each language.
hFetchProperties	Used to set the properties of the fetch (note that some properties like for instance URL_BASE are also used for file fetching). The properties are stored in a map; the functions TtsMapCreate, TtsMapDestroy, TtsMapSetChar, TtsGetChar etc. are used to maintain the map. The list of available properties can be found in the header file “lh_inettypes.h” (for example SPIINET_URL_BASE to support relative URI's and filenames and SPIINET_TIMEOUT_DOWNLOAD to set the fetch timeout.) It's optional: specify NULL if not used.

SpeakData (PSpeakData)

SpeakData is used when calling TtsProcessEx. It is used to describe the location of the input data for a text to speech action and its properties. Note that it is still possible to use the source call-back method; in this case the *uri* and *data* structure members should be set to NULL.
PSpeakData is a pointer to a SpeakData structure.

```
typedef struct
```

```
{
char*           uri;
VOID*           data;
U32             lengthBytes;
char*           contentType;
HTTSMAP         fetchProperties;
HTTSVECTOR      fetchCookieJar;
} SpeakData, *PSpeakData;
```

Structure members

uri	String specifying the location of the input data. This can be an http address (http://)or a file name (regular or with file://). Since file context can be interpreted based on the file extensions, file extensions are required. The following file extensions are interpreted and the file content is read as defined in the following table (set contentType to NULL unless you want to overwrite this behavior):

Extension	Content read as
.txt	Text file
.xml	4sml file
.ssml	4sml file

Set the uri member to NULL to indicate that the input data is provided via the *data* member or the source call-back.

data	Pointer to buffer containing the input text. This structure member will be used only when uri is NULL. Set both uri and data to NULL to use the source callback function.
lengthBytes	The length of the data buffer in bytes.

Member	Description
contentType	String that describes the type of content of data. This string must be specified when a file with unsupported file extension is included in the uri string or when using the data member: for more info see the description of the uri member. contentType can have the following values. Remark: the string values are case-sensitive; capitals are not supported.

Value	Content
“default”	Text file
“text/plain”	Text file.
“text/plain;charset= <charset>”	Text file. It’s optional to add a specification of the character set. Example: “text/plain;charset= windows-1252” See the table below for an overview of some of the supported character sets.
“application/synthesis+ssml”	4sml file.
“application/ssml+xml”	4sml file.
“text/xml”	4sml file.

Member	Description
fetchProperties	Used to set the properties of the fetch. The properties are stored in a map; the functions TtsMapCreate, TtsMapDestroy, TtsMapSetChar, TtsMapGetChar etc. are available to manipulate a map. The list of available properties can be found in the header file lh_inettypes.h
fetchCookieJar	For internal use.

The supported abstract character set varies by language. But note that an abstract character set is distinct from a coded character set. In fact RealSpeak supports all coded character sets supported by the ICU transcoding component for all languages as long as the input text only contains characters that can be transcoded to the native coded character set of the language of the input. See the "RealSpeak languages" appendix for a list of the native character set for each language.
Some examples are listed below.

Coded Character set	**Languages**	**Notes**
UTF-8	All languages	
UTF-16	All languages	If the byte order mark is missing, big-endian is assumed.
ISO-8859-1	Western languages	
windows-1252	Western languages	
EUC-jp (synonym: EUC)	Japanese	
Shift-JIS	Japanese	

The ICU component which is used to perform the transcoding from the input character set to the native character set is a third-party component. For more information see also the "Copyright and Licensing for third party software" appendix.

For more information about the character sets for the contentType parameter, take a look at the following websites:

- The ICU website: http://www-306.ibm.com/software/globalization/icu
- www.iana.org/assignments/character-sets

Note that in fact RealSpeak supports all character sets supported by the ICU component for all languages as long as the input text only contains characters that can be transcoded to the native character set of the language of the input. For example an input text

Remark:

The application cannot specify the contenttype/charset argument when using the TTSSOURCECB source callback mechanism: the character set is then chosen by the TTS system based on the active language; see the "RealSpeak languages" appendix for the native character set for each language. In that case, the application has to make sure that the input text character set matches this character set.

DictionaryData, (PDictionaryData)

DictionaryData is used when calling TtsLoadUsrDictEx, TtsUnloadUsrDictEx or TtsEnableUsrDictEx(). It is used to describe the properties of a dictionary instance. PDictionaryData is a pointer to a DictionaryData structure.

```
typedef struct
{
U32             version;
char*           uri;
VOID*           data;
U32             lengthBytes;
char*           contentType;
HTTSMAP         fetchProperties;
HTTSVECTOR      fetchCookieJar;
} DictionaryData, *PDictionaryData;
```

Structure members

version	Dictionary version.
uri	String specifying the location of a dictionary this can be an http address (http://)or a file name (regular or with file://). Since file context can be interpreted based on the file extensions, file extensions are required. The following file extensions are interpreted and the file content is read as defined in the following table (set contentType to NULL unless you want to overwrite this behavior):

Value	Content read as
.dct	Text file.
.tdc	Text file.
.bct	Binary file.
.tbc	Binary file.

Set the uri member to NULL to indicate that the input data is read from the data member.

data	Pointer to data stream. This structure member will be used when uri is NULL.
lengthBytes	The length of the data stream in bytes.
contentType	String that describes the type of content of data. This string must be specified when a file with unsupported file extension is specified by the uri structure member or when working with the data component.

contentType can have the following constants as value:

Value	File type
DCT_MIME_EDCT_TEXT	Text
DCT_MIME_EDCT	Binary

fetchProperties	Used to set the properties of the fetch if an URL is specified. The properties are stored in a map; the functions TtsMapCreate, TtsMapDestroy, TtsMapSetChar, TtsGetChar etc. are used to maintain the map. The list of available properties can be found in the header file lh_inettypes.h
fetchCookieJar	For internal use.

G2P_DICTNAME

G2P_DICTNAME is used to represent a G2P dictionary.

```
typedef char
G2P_DICTNAME[MAX_G2P_DICTNAME_LENGTH]
```

TTS_Marker

TTS_Marker provides bit masks for all marker types. By bitwise or'ing the types of interest, an integer is created that can be used to specify the TTS_MARKER_MODE_PARAM parameter via the TtsSetParam(s) functions. Only the corresponding event types will be issued by the event callback function. However, SENTENCEMARK events are always generated.

```
typedef enum TTS_Marker
{
TTS_MRK_SENTENCE           = 0x0001,
TTS_MRK_WORD               = 0x0002,
TTS_MRK_PHONEME            = 0x0004,
TTS_MRK_BOOK               = 0x0008,
TTS_MRK_PARAGRAPH          = 0x0200
} TTS_Marker;
```

TTS_Event

TTS_Event defines all event types that can be caught by the TTSEVENTCB callback function.

```
typedef enum TTS_Event
{
TTS_EVENT_SENTENCEMARK,
TTS_EVENT_BOOKMARK,
TTS_EVENT_WORDMARK,
TTS_EVENT_PHONEMEMARK,
TTS_EVENT_PARAGRAPHMARK,
} TTS_Event;
```

Events normally mark the beginning of a particular kind of data (sentence, word…) in the audio output. But an event will also be issued when the audio output reaches the position of a book mark inserted in the input text.

TTS_EVENT_BOOKMARK	Marks the position of a user book mark; book marks can be inserted in the input text via the SSML <mark> element or the RealSpeak <esc>\mrk=x\ tag. The type TTS_BookMark is be used to store the marker's properties.
TTS_EVENT_SENTENCEMARK	Marks the beginning of a sentence. The type TTS_SentenceMark is be used to store the marker's properties.
TTS_EVENT_WORDMARK	Marks the beginning of a word. The type TTS_WordMark is

	used to store the marker’s properties.
TTS_EVENT_PARAGRAPHMARK	Marks the beginning of a paragraph. The type TTS_ParagraphMark is be used to store the marker’s properties. Note that paragraph markers are only issued when paragraphs have been marked in the input text via the paragraph tag (native <ESC>\p\ tag or SSML <p> element).
TTS_EVENT_PHONEMEMARK	Marks the beginning of a phoneme. The type TTS_PhonemeMark is be used to store the marker’s properties.

TTS_MarkPos

TTS_MarkPos describes the common properties of a marker. This structure is part of a marker structure that describes a particular kind of marker (TTS_BookMark, TTS_PhonemeMark…).

```
typedef struct TTS_MarkPos {
{
U32                              nInputPos;
U32                              nInputLen;
U32                              nOutputPos;
U32                              nOutputLen;
} TTS_MarkPos;
```

Structure members

nInputPos	Start position of input (sentence, word…) in bytes. Start position is counted from beginning of message.
nInputLen	Length of input (sentence, word...) in bytes.
nOutputPos	Start position of output (sentence, word..) in bytes. Start position is counted from beginning of message.
nOutputLen	Length of output (sentence, word...) in bytes.

Not every marker type supports the four attributes. Here's an overview of which TTS_EVENT event type is supporting what kind of data:

Event type (excl. TTS_EVENT prefix)	**nInput Pos**	**nInput Len**	**nOutput Pos**	**nOutput Len**
BOOKMARK	Yes	No	Yes	No
SENTENCEMARK	Yes	Yes	Yes	No
WORDMARK	Yes	Yes	Yes	No
PARAGRAPH MARK	Yes	No	Yes	No
PHONEMEMARK	No	No	Yes	Yes

TTS_BookMark

TTS_BookMark describes the parameters for a bookmark marker. This structure is passed to TTSEVENTCB callback when the event is TTS_EVENT_BOOKMARK.
It is important to note that RealSpeak Telecom 4.0 only supports numerical bookmarks.

```
typedef struct TTS_BookMark {
  const char *            szID;
  TTS_MarkPos             mrkPos;
} TTS_BookMark;
```

TTS_PhonemeMark

TTS_PhonemeMark describes the parameters for a phoneme marker. This structure is passed to TTSEVENTCB callback when the event is TTS_EVENT_PHONEMEMARK.

```
typedef struct TTS_PhonemeMark {
  const char *                    szID;
  TTS_MarkPos                     mrkPos;
} TTS_PhonemeMark;
```

TTS_SentenceMark

TTS_SentenceMark describes the parameters for a sentence marker. This structure is passed to TTSEVENTCB callback when the event is TTS_EVENT_SENTENCEMARK.

```
typedef struct TTS_SentenceMark {
  TTS_MarkPos                     mrkPos;
} TTS_BookMark;
```

TTS_ParagraphMark

TTS_ParagraphMark describes the parameters for a paragraph marker. This structure is passed to TTSEVENTCB callback when the event is TTS_EVENT_PARAGRAPHMARK.

```
typedef struct TTS_ParagraphMark {
  TTS_MarkPos                     mrkPos;
} TTS_ParagraphMark;
```

TTS_WordMark

TTS_WordMark describes the parameters for a word marker. This structure is passed to TTSEVENTCB callback when the event is TTS_EVENT_WORDMARK.

```
typedef struct TTS_WordMark {
  TTS_MarkPos                    mrkPos;
} TTS_WordMark;
```

Structure members

All marker structures have the following structure members in common:

szID	Currently not used (for Bookmark and Phoneme markers only)
mrkPos	Struct that describes the data values for one marker. Consists of nInputLen, nInputValue, nOutputLen and nOutputValue.

Function Descriptions

TtsInitializeEx

Syntax:

```
TTSRETVAL TtsInitializeEx(
                    HTTSINSTANCE* tts_handle,
                    LH_SDK_SERVER* server,
                    TTSPARM* lpTtsParms,
                    void* lpAppData)
```

Purpose: Initializes an instance of the TTS engine instance on the server specified by the server parameter. The engine is created according to the information specified in the lpTtsParms parameter. When operating in local or in-process mode, pass NULL for the server parameter.

To create an instance in Client/Server mode, you must call TtsCreateEngine before calling TtsInitializeEx.

Parameters:

tts_handle	[out] This pointer receives the handle for the newly created TTS engine instance. This handle is passed to other TTS functions to identify the instance
server	Pointer to a server information structure, which has been initialized by a call to CreateEngine(Set to NULL if not in client/server mode)
lpTtsParms	This is a pointer to a structure whose members are used to control certain aspects and behaviors of the created engine. Some members of this struct **must** be filled in; check the TTSPARM struct information for more details.
lpAppData	Application specific data that is passed back to the application each time one of the callbacks is invoked.

Error codes: This function can return license related errors. See TtsResourceAllocate() for more info.

TtsInitialize

Syntax:

```
TTSRETVAL TtsInitialize(
                HTTSINSTANCE* phTtsInst,
                LH_SDK_SERVER* pServer,
                TTSPARM* pTtsParms,
                void* pAppData)
```

Purpose: Initializes an instance of the TTS engine instance on the server specified by the pServer parameter. The engine is created according to the information specified in the pTtsParms parameter. When operating in local or in-process mode, pass NULL for the pServer parameter.

To create an instance in Client/Server mode, you should call TtsCreateEngine before calling TtsInitialize.

Parameters:

phTtsInst	[out] This pointer receives the handle for the newly created TTS engine instance. This handle is passed to other TTS functions to identify the instance
pServer	Pointer to a server information structure, which has been initialized by a call to CreateEngine(Set to NULL if not in client/server mode)
pTtsParms	This is a pointer to a structure whose members are used to control certain aspects and behaviors of the created engine.
pAppData	Application specific data that is passed back to the application each time one of the callbacks is invoked.

TtsUninitialize

Syntax:
TTSRETVAL TtsUninitialize(HTTSINSTANCE hTtsInst)

Purpose: Frees all system resources associated with an engine instance. Note that this function does not unload user dictionaries; that must be done using TtsUnloadUsrDict.

Parameters:

hTtsInst	Handle to a TTS engine instance

Error codes: This function can return license related errors. See TtsResourceFree() for more info.

TtsProcessEx

Syntax:

```
TTSRETVAL TtsProcessEx (
                    HTTSINSTANCE tts_handle,
                    const SpeakData * pSpeakData)
```

Purpose: Convert input text data into speech of the previously specified output format. The input data properties can be described via the pSpeakData argument. To use the source call-back, set the structure members specifying the input (*uri* and *data*) to NULL.

Parameters:

hTtsInst	Handle to a TTS engine instance
pSpeakData	Pointer to SpeakData typed structure, which describes where the input data for text to speech can be found and its properties. Refer to the SpeakData structure type description for more details.

Error codes: This function can return licensing related errors. See TtsResourceAllocate() for more info.

Remark: When using the TtsProcessEx function, the TTS input method is determined as follows: first the *uri* SpeakData structure member is checked: if it's non-NULL the specified URI is used, else the *data* member is checked: if it's non-NULL, the specified buffer is used. Only when both the *uri* and the *data* member are NULL, the source callback function of type TTSSOURCECB is used.
Refer to the description of the SpeakData structure for more details about the supported character sets for the input text data.

TtsProcess

Syntax:

```
TTSRETVAL TtsProcess (HTTSINSTANCE hTtsInst)
```

Purpose: Convert input text data into speech of the previously specified output format. The input data is received from the application through the TTSSOURCECB callback. The speech data is delivered using the TTSDESTCB destination callback. The function returns either when all the speech data has been delivered or when the data delivery has been stopped by a call to TtsStop. The format of the output data can be specified when initializing the engine, and the speech can be tuned via calls to TtsSetParam (e.g. set the volume, the rate, the voice).

Parameters:

hTtsInst	Handle to a TTS engine instance

The input text retrieved via the source callback must be encoded in the native character set for the active language. Refer to the "RealSpeak Languages" appendix for the native character set for each language.

TtsStop

Syntax:

```
TTSRETVAL TtsStop(HTTSINSTANCE hTtsInst)
```

Purpose: Stops the Text-To-Speech conversion process initiated by a call to TtsProcess() or TtsProcessEx(). Since the process functions are synchronous (blocking), the TtsStop function must be called from a different thread than the one that called the TtsProcess(Ex) function. The TtsStop function always succeeds unless the engine instance is not speaking.

Parameters:

hTtsInst	Handle to a TTS engine instance

TtsSetParam

Syntax:

```
TTSRETVAL TtsSetParam(
                HTTSINSTANCE hTtsInst,
                U16 nParam,
                U16 nValue
```

Purpose: Sets a TTS engine instance parameter to a specified value. When the parameter change will take effect depends on the parameter that is being set; when setting TTS_VOLUME_PARAM or TTS_RATE_PARAM, the change will take effect almost immediately. This function may be called while the TtsProcess(Ex) function is processing, but some parameters cannot be changed while TtsProcess(Ex) is active (e.g. language, voice, document type, markup type, output type, marker mode).

If you want to set more than one parameter in one go, and one of the parameters to be set is the document type, always set the document type first before setting the other parameters. Otherwise, the change to the other parameter might fail.

Notes:

- When using e-mail preprocessing, word-by-word and line-by-line read mode is not available
- In a client/server environment, the default rate and volume is set in RealSpeak Server configuration file (see "Configuration Files" section of "User Configuration" chapter). If the rate or volume is set through this API call, the new value overrides those defaults. Similarly, if the rate or volume is set through markup in the input text, those values override both the RealSpeak Server default and the value set via the API for that (and only that) speak request.

Parameters:

hTtsInst	Handle to a TTS engine instance
nParam	Parameter to set
nValue	Value to set

On the next page is a table of currently supported parameters and some of their corresponding default values:

Parameter	Acceptable Values	Default Value
TTS_LANGUAGE_PARAM	ASCII character string value stored in the paramValue.array field; language name.	
TTS_VOICE_PARAM	ASCII character string value stored in paramValue.array field; voice name. Be careful to use an existing voice name/language combination!	
TTS_VOLUME_PARAM	0 to 9 (inclusive)	8
TTS_RATE_PARAM	1 to 9 (inclusive)	5
TTS_DOCUMENT_TYPE_PARAM	DOC_NORMAL, DOC_EMAIL	DOC_NORMAL
TTS_VOLUME_LARGESCALE_PARAM	0 to 100 (inclusive)	80
TTS_RATE_LARGESCALE_PARAM	1 to 100 (inclusive)	50
TTS_MARKUP_TYPE_PARAM	MARKUP_NONE, MARKUP_4SML	MARKUP_NONE
TTS_OUTPUT_TYPE_PARAM	0, 1, 2	
TTS_MARKER_MODE_PARAM		TTS_MRK_SENTENCE | TTS_MRK_BOOK | TTS_MRK_PARAGRAPH
TTS_BLADE_ENABLE_PARAM	Currently not used	
TTS_BLADE_DISABLE	Currently not used	
TTS_LICENSE_MODE_PARAM	LICENSE_MODE_DEFAULT, LICENSE_MODE_EXPLICIT	LICENSE_MODE_DEFAULT
TTS_RULESET_LOAD_PARAM[1]	Pointer to TTS_FETCHINFO_T structure stored in paramValue.pObj field (see "User Rulesets" section in the "User Configuration" chapter for more details)	No rulesets are loaded
TTS_RULESET_UNLOAD_PARAM[2]	Pointer to TTS_FETCHINFO_	

[1] Note that TTS_RULESET_LOAD_PARAM and TTS_RULESET_UNLOAD_PARAM are parameters that can only be specified in TtsSetParam or TtsSetParams call and never in a TtsGetParam or TtsGetParams call.

	T structure stored in paramValue.pObj field (see "User Rulesets" section in the "User Configuration" chapter for more details)	

[2] See the previous footnote.

TtsGetParam

Syntax:

```
TTSRETVAL TtsGetParam(
              HTTSINSTANCE hTtsInst,
              U16 nParam,
              U16* pnValue)
```

Purpose: Gets the value of a given parameter. See TtsSetParam for a list of supported parameters and possible values.

The TtsGetParam and TtsSetParam functions operate independently of the escape sequences that can also be used to set the volume and rate. Calls to TtsGetParam will not reflect parameter changes that result from escape sequences embedded in the input text. The effect of playing pre-processed email text will also not be reflected in the values returned by the TtsGetParam function.

Parameters:

hTtsInst	Handle to a TTS engine instance
nParam	Specifies which parameter value to retrieve
nValue	[out] Current value of nParam

TtsSetParams

Syntax:

```
TTSRETVAL TtsSetParams(
            HTTSINSTANCE hTtsInst,
            TTS_PARAM_T*  pParamList,
            U16  nParamNb)
```

Purpose: Sets one or more TTS engine instance parameters to a specified value. When the parameter change will take effect depends on the parameter that is being set; when setting TTS_VOLUME_PARAM or TTS_RATE_PARAM, the change will take effect almost immediately. This function may be called while the TtsProcess(Ex) function is processing, but some parameters cannot be changed while TtsProcess(Ex) is active (e.g. language, voice, document type, markup type, output type, marker mode).

Notes:

- See TtsSetParam note on the default rate and volume in a client/server environment.

Parameters:

hTtsInst	Handle to a TTS engine instance
pParamList	List of parameters to set
nParamNb	Number of parameters to set

For a table of currently supported parameters and their corresponding values, see TtsSetParam.

TtsGetParams

Syntax:

```
TTSRETVAL TtsGetParams(
              HTTSINSTANCE hTtsInst,
              TTS_PARAM_T*  pParamList,
              U16  nParamNb)
```

Purpose: Gets the values of given parameters. See TtsSetParam for a list of supported parameters and possible values.

The TtsGetParam(s) and TtsSetParam(s) functions operate independently of the escape sequences that can also be used to set the volume and rate. Calls to TtsGetParams will not reflect parameter changes that result from escape sequences embedded in the input text. The effect of playing pre-processed email text will also not be reflected in the values returned by the TtsGetParam function.

Parameters:

hTtsInst	Handle to a TTS engine instance
pParamList	Specifies which parameter values to retrieve
nParamNb	Number of parameters to retrieve

TtsLoadUsrDictEx

Syntax:

```
TTSRETVAL TtsLoadUsrDictEx (
            HTTSINSTANCE hTtsInst,
            const DictionaryData* dictionary,
            HTTSDCTEG* phDctEg)
```

Purpose: Loads a user dictionary instance into memory. The dictionary is implicitly enabled with the default priority (see TtsEnableUsrDictEx for making dictionaries explicitly enabled with a chosen priority). In Client/Server mode, the dictionary is loaded on the server. The file format of a dictionary is described in the "User Configuration" chapter.

Each dictionary instance is initialized with the default (lowest) priority. All dictionary instances must have a different priority (except the default priority, which can be used by several dictionaries). The priority can be set by TtsEnableUsrDictEx() If two dictionaries have the default priority, the order in which the dictionaries are loaded is important. The last loaded dictionary has the 'highest' priority. This means that when a token has to be processed, a lookup will take place using the last loaded dictionary first.

Parameters:

hTtsInst	Handle to a TTS engine instance
dictionary	Const pointer to a dictionary properties description.
phDctEg	[out] Handle to a dictionary instance.

The order in which dictionaries are looked up can be changed by using TtsEnableUsrDictEx() and setting the priority to a different value.

Remark

The dictionary is implicitly made enable for use with the default priority; this behavior is different compared to TtsLoadUsrDict.

Refer to "User Configuration" chapter for more info.

TtsLoadUsrDict

Syntax:

```
TTSRETVAL TtsLoadUsrDict (
        LH_SERVER_INFO* pServer,
        HTTSDICT* phUsrDict,
        char* szUserDict)
```

Purpose: Loads a user dictionary instance into memory. In order to be used by a TTS engine instance, the dictionary must be enabled for that instance by calling the TtsEnableUsrDict function. In Client/Server mode, the dictionary is loaded on the server specified by pServer. In local mode, the pServer parameter should be set to NULL. The dictionary is a file whose format is described in "User Configuration" chapter.

Parameters:

pServer	Pointer to a server information structure. To open a local dictionary, set this parameter to NULL
phUsrDict	[out] Handle to the loaded dictionary instance.
szUsrDict	Fully qualified pathname to the user dictionary that will be loaded

Remark

The dictionary is only loaded into memory and not made implicit enabled; this behavior is different compared to TtsLoadUsrDictEx.

TtsUnloadUsrDictEx

Syntax:

```
TTSRETVAL TtsUnloadUsrDictEx(
        HTTSINSTANCE hTtsInst,
        HTTSDCTEG hDctEg)
```

Purpose: Unloads a user dictionary instance, freeing the resources associated with it. As opposed to TtsUnloadUsrDict, a user dictionary instance can be unloaded when it is enabled; a user dictionary is either enabled implicitly by TtsLoadUsrDictEx or explicitly by TtsEnableUsrDictEx.

Parameters:

hTtsInst	Handle to a TTS engine instance.
hDctEg	Handle to a dictionary instance.

Refer to "User Configuration" chapter for more info.

TtsUnloadUsrDict

Syntax:

```
TTSRETVAL TtsUnloadUsrDict(HTTSDICT hUsrDict)
```

Purpose: Unloads a user dictionary, freeing the resources associated with it. This function will fail if the dictionary is enabled by an engine instance. If the dictionary is enabled then call TtsDisableUsrDict to disable the dictionary.

Parameters:

hUsrDict	Handle to the loaded dictionary that will be unloaded

TtsEnableUsrDictEx

Syntax:

```
TTSRETVAL TtsEnableUsrDictEx(
        HTTSINSTANCE hTtsInst,
        HTTSDCTEG hDctEg,
        U32 priority)
```

Purpose: Enables a user dictionary instance and/or changes its priority on a TTS engine instance.
Once a dictionary instance has been loaded by TtsLoadUsrDictEx() on a TTS engine instance, the default priority has been attached; the dictionary is enabled with default (lowest) priority.
To change the default priority, the dictionary has to be disabled by calling TtsDisableUsrDictEx and enabled again by calling TtsEnableUsrDictEx().
TtsEnableUsrDictEx() can also be called to enable a dictionary again that has been disabled by a previous call of TtsDisableUsrDictEx().
If a dictionary instance has been opened in Client/Server mode then it can only be enabled for an instance that was created on the same server as the dictionary. Once a dictionary has been loaded using the TtsLoadUsrDictEx function, it can be enabled for use by only one TTS engine instance at a time; if two instances want to use the same dictionary then the dictionary must be loaded separately for each instance. Each dictionary has a unique priority; no two dictionaries can have the same priority at the same time except the default priority. The highest possible priority value is 0; the higher the priority, the lower the value of the priority parameter should be.

Parameters:

hTtsInst	Handle to a TTS engine instance.
hDctEg	Handle to a loaded dictionary instance
priority	Sets the priority for the dictionary instance.

Remark:

Always call TtsDisableUsrDict(s)Ex before calling TtsEnableUsrDictEx.

Refer to “User Configuration” chapter for more info.

TtsEnableUsrDict

Syntax:

```
TTSRETVAL TtsEnableUsrDict(
        HTTSINSTANCE hTtsInst,
        HTTSDICT hUsrDict)
```

Purpose: Enables a user dictionary on a TTS engine instance. If a dictionary has been opened in Client/Server mode, it can only be enabled for an instance that was created on the same server as the dictionary. Once a dictionary has been loaded using the TtsLoadUsrDct function, it can be enabled for use by only one engine instance at a time. If two instances want to use the same dictionary then the dictionary must be loaded and enabled separately for each instance.

Parameters:

hTtsInst	Handle to a TTS engine instance.
hUsrDict	Handle to a loaded dictionary instance

TtsDisableUsrDictEx

Syntax:

```
TTSRETVAL TtsDisableUsrDictEx(
                HTTSINSTANCE hTtsInst,
                HTTSDCTEG hDctEg)
```

Purpose: Disables a user dictionary instance on a TTS engine instance. The dictionary instance must first have been enabled for use by the instance using TtsEnableUsrDictEx or TtsLoadUsrDictEx. Note that disabling the dictionary does not unload it from memory. To unload a dictionary, use the TtsUnloadUsrDictEx function.

Parameters:

hTtsInst	Handle to a TTS engine instance.
hDctEg	Handle to a loaded dictionary instance

Remark:

Always use this function before calling TtsEnableUsrDictEx.

Refer to “User Configuration” chapter for more info.

TtsDisableUsrDict

Syntax:

```
TTSRETVAL TtsDisableUsrDict(
                HTTSINSTANCE hTtsInst,
                HTTSDICT hUsrDict)
```

Purpose: Disables a user dictionary instance on a TTS engine instance. The dictionary must first have been enabled for use by the instance using TtsEnableUsrDict. Note that disabling the dictionary does not unload it from memory. To unload a dictionary, use the TtsUnloadUsrDict function.

Parameters:

hTtsInst	Handle to a TTS engine instance.
hUsrDict	Handle to a loaded dictionary instance

TtsDisableUsrDictsEx

Syntax:

```
TTSRETVAL TtsDisableUsrDictsEx(HTTSINSTANCE hTtsInst
```

Purpose: Disables all user dictionary instances on a TTS engine instance. The dictionary instances must first have been enabled for use by the instance using TtsEnableUsrDictEx or TtsLoadUsrDictEx. Note that disabling the dictionaries does not unload them from memory. To unload a dictionary, use the TtsUnloadUsrDictEx function. To disable dictionaries one by one use TtsDisableUsrDictEx.

Parameters:

hTtsInst	Handle to a TTS engine instance.

Refer to "User Configuration" chapter for more info.

TtsLoadG2PDictList

Syntax:

```
TTSRETVAL TtsLoadG2PDictList (
            HTTSINSTANCE hTtsInst,
            U32 u32NumDictNames,
            G2P_DICTNAME* pG2PDictList)
```

Purpose: Loads a list of custom G2P dictionaries on a TTS engine instance.

Parameters:

hTtsInst	Handle to a TTS engine instance.
u32NumDictNames	Number of names in the array
pG2PDictList	Pointer to an array of names to enable

TtsUnloadG2PDictList

Syntax:

```
TTSRETVAL TtsUnloadG2PDictList (
            HTTSINSTANCE hTtsInst,
            U32 u32NumDictNames,
            G2P_DICTNAME* pG2PDictList)
```

Purpose: Unloads a list of custom G2P dictionaries on a TTS engine instance.

Parameters:

hTtsInst	Handle to a TTS engine instance.
u32NumDictNames	Number of names in the array
pG2PDictList	Pointer to an array of names to disable

TtsGetG2PDictTotal

Syntax:

```
TTSRETVAL TtsGetG2PDictTotal (
            HTTSINSTANCE hTtsInst,
            U32* pu32Total)
```

Purpose: Retrieves the total number of custom G2P dictionaries on the system for the current language. This allows the user to allocate appropriate memory for calling TtsGetLoadedG2PList.

Parameters:

hTtsInst	Handle to a TTS engine instance.
pu32Total	[out] Total number of G2P dictionaries

TtsGetG2PDictList

Syntax:

```
TTSRETVAL TtsGetG2PDictList (
            HTTSINSTANCE hTtsInst,
            U32 u32NumAllocated,
            G2P_DICTNAME* pG2PDictList,
            U32* pu32NumRetreived)
```

Purpose: Gets the list of custom G2P dictionaries on the system for the current language. Before calling this method, the user must call TtsGetG2PDictTotal and allocate enough memory for the list.

Parameters:

hTtsInst	Handle to a TTS engine instance.
u32NumAllocated	Number of G2P_DICTNAMEs that can fit in the list
pG2PDictList	[out] Pointer to a previously allocated array of G2P dictionary names that will be filled
pu32NumRetreived	[out] Number of names that were actually put in the list

TtsMapCreate

Syntax:

TTSRETVAL TtsMapCreate(HTTSMAP* phTtsMap)

Purpose: Create an empty map that can be used to store the fetch properties specified in the SpeakData, DictData. or TTS_FETCHINFO_T structures

Parameters:

phTtsMap	Pointer to a handle to a TTS Map instance.

TtsMapDestroy

Syntax:

```
TTSRETVAL TtsMapDestroy(HTTSMAP hTtsMap)
```

Purpose: Destroys a map.

Parameters:

hTtsMap	Handle to a TTS Map instance.

TtsMapSetChar

Syntax:

```
TTSRETVAL TtsMapSetChar (
                    HTTSMAP hTtsMap,
                    const char* szKey,
                    const char* szValue)
```

Purpose: Set a named property of type string (char *) on a map. The list of available properties can be found in the header file lh_inettypes.h.

Parameters:

hTsMap	Handle to a TTS Map instance.
szKey	The name of the property.
szValue	The value of the property

TtsMapSetU32

Syntax:

```
TTSRETVAL TtsMapSetU32 (
            HTTSMAP hTtsMap,
            const char* szKey,
            U32 nValue)
```

Set a named property of type unsigned 32-bit integer (U32) on a map. The list of available properties can be found in the header file lh_inettypes.h.

Parameters:

hTsMap	Handle to a TTS Map instance.
szKey	The name of the property.
nValue	The value of the property

TtsMapSetBool

Syntax:

```
TTSRETVAL TtsMapSetBool (
                    HTTSMAP hTtsMap,
                    const char* szKey,
                    int bValue)
```

Purpose: Set a named property of type boolean (int) on a map. The list of available properties can be found in the header file lh_inettypes.h.

Parameters:

hTsMap	Handle to a TTS Map instance.
szKey	The name of the property.
bValue	The value of the property

TtsMapGetChar

Syntax:

```
TTSRETVAL TtsMapGetChar (
            HTTSMAP hTtsMap,
            const char* szKey,
            char** pszValue)
```

Purpose: Gets a named property of type string (char *) from a map. TtsMapGetChar is responsible for the memory allocation of the string.

Parameters:

hTsMap	Handle to a TTS Map instance.
szKey	The name of the property.
pszValue	*pszValue The value of the property

TtsMapFreeChar

Syntax:

```
TTSRETVAL TtsMapFreeChar (
            HTTSMAP hTtsMap,
            char* szValue)
```

Purpose: clean up the allocated memory in use by szValue. This memory has been allocated by TtsGetMapChar .

Parameters:

hTsMap	Handle to a TTS Map instance.
szValue	The value of the property

TtsMapGetU32

Syntax:

```
TTSRETVAL TtsMapGetU32 (
            HTTSMAP hTtsMap,
            const char* szKey,
            U32* pnValue)
```

Purpose: Gets a named property from a map. The value of the property is a U32.

Parameters:

hTsMap	Handle to a TTS Map instance.
szKey	The name of the property.
pnValue	The value of the property

TtsMapGetBool

Syntax:

```
TTSRETVAL TtsMapGetBool (
              HTTSMAP hTtsMap,
              const char* szKey,
              int* pbValue)
```

Purpose: Gets a property from a map. The value of the property is a boolean.

Parameters:

HTsMap	Handle to a TTS Map instance.
SzKey	The name of the property.
PbValue	The value of the property

TtsCreateEngine

Syntax:

```
TTSRETVAL TtsCreateEngine(LH_SDK_SERVER* pServer)
```

Purpose: Used only for Client/Server mode. In RealSpeak v3.5 this function created an engine instance on the server. Currently the creation of the engine instance is deferred until TtsInitialize( ) is called; making this function essentially a NO-OP that is only available for backward compatibility.

This function should only be called once for each engine instance to be created. The LH_SERVER_INFO member of the LH_SDK_SERVER data structure is used to specify the network information necessary to connect to the server. The other member of the LH_SERVER_INFO data structure is a handle to the created engine instance, which is filled in by the function. The fully initialized structure is then passed to the TtsInitialize(Ex) function.

If the server resides locally, the IP address can be set to 127.0.0.1 or 'localhost' to specify the local host.

Parameters:

pServer	[in/out]Pointer to a server information structure

TtsRemoveEngine

Syntax:

```
TTSRETVAL TtsRemoveEngine(LH_SDK_SERVER* pServer)
```

Purpose: Used only for Client/Server mode. It removes an engine instance from the server. When closing an engine instance, you should first call the TtsUninitialize function to clean up engine instance data.

Parameters:

pServer	Pointer to a server information structure

TtsResourceAllocate

Syntax:

```
TTSRETVAL TtsResourceAllocate (
              HTTSINSTANCE hTtsInst,
              const char*  szFeature,
              void*  pReserved
```

Purpose:

Explicitly retrieve a license from the license server for a specified RealSpeak instance.

Parameters

hTtsInst	Handle to a TTS engine instance
szFeature	The function is generic: use the constant TTS_LICENSE_SPEAK for licensing functionality.
reserved	This parameter is reserved for future use. Pass in NULL

Notes

The TTS_LICENSE_MODE_PARAM parameter must be set to 'explicit' for TtsResourceAllocate( ) to work. You can use TtsGetParam( ) to retrieve the value of TTS_LICENSE_MODE_PARAM and find out whether you need to call this function (and explicitly allocate and free licenses) or not. If the licensing mode is set to "default," the TtsInitializeEx( ) function implicitly allocates a license for the TTS engine instance and TtsUnintialize() releases that license.

TtsResourceAllocate( ) may return the following error codes:

TTS_E_INVALIDPARAM	An invalid feature parameter was specified
TTS_E_LIC_LICENSE_ALLOCATED	A license has already been allocated for this TTS engine instance.
TTS_E_WRONG_STATE	A speak operation is active
TTS_E_LIC_NO_LICENSE	There are no purchased licenses available
TTS_E_LIC_UNSUPPORTED	The TTS_LICENSE_MODE_PARAM parameter is not set to explicit.

See also

TtsResourceFree( )

TtsResourceFree

Syntax:

```
TTSRETVAL TtsResourceFree (
                HTTSINSTANCE hTtsInst,
                const char*  szFeature,
                void*        reserved)
```

Purpose: Explicitly free the license for the specified Realspeak instance.

Parameters

hTtsInst	Handle to a TTS engine instance
szFeature	The function is generic: use TTS_LICENSE_SPEAK to free a license
reserved	This parameter is reserved for future use. Pass in NULL.

Notes

The TTS_LICENSE_MODE_PARAM parameter must be set to 'explicit' for TtsResourceFree( ) to work.

TtsResourceFree( ) may return the following error codes:

TTS_E_INVALIDPARAM	An invalid feature parameter was specified
TTS_E_LIC_LICENSE_FREED	A license has already been freed for this TTS engine instance.
TTS_E_WRONG_STATE	A speak operation is active.
TTS_E_LIC_NO_LICENSE	There are no purchased licenses available
TTS_E_LIC_UNSUPPORTED	The TTS_LICENSE_MODE_PARAM parameter is not set to explicit

See also

TtsResourceAllocate( )

User Callbacks

This section describes the callbacks that the user (application) needs to implement and register when using the RealSpeak SDK. Callbacks are registered by passing their pointers in the TTSCallbacks typed structure specified via the *cbFuncs* TTSPARM member when TtsInitialize or TtsInitializeEx is used..

TTSSOURCECB

```
Typedef TTSRETVAL (*TTSSOURCE)(
                        void* pAppData,
                        void* pDataBuffer,
                        U32 nBufferSize,
                        U32* pnDataSize);
```

Purpose: This callback is only invoked when the input streaming mode of the TTS engine is enabled.
It is used by an engine instance to request a block of input text from the application. The function is called multiple times, allowing an unlimited amount of data to be delivered.
Each time the application puts data into *pDataBuffer*, the function should return TTS_SUCCESS.
When there is no more input data for the current TTS action the function should return TTS_ENDOFDATA. Then, the TTS engine knows the previous input corresponded with the last input block for a Speak action and the call-back will no longer be called until the TtsProcess(Ex) function returns. Any data in the buffer when TTS_ENDOFDATA is returned is ignored.

Parameters:

pAppData	Application data pointer that was passed into TtsInitialize
pDataBuffer	[out] Pointer to a data buffer that is to be filled with the input text. This buffer is provided by the callback function, no need to allocate memory for it.
nBufferSize	Size in bytes of the buffer pointed to by pDataBuffer. This is the maximum amount of data that can be placed in pDataBuffer
pnDataSize	[out] Number of bytes that were actually placed in the buffer

Return Values:

TTS_SUCCESS
TTS_ENDOFDATA

Remark: When using TtsInitializeEx, this callback should not be registered unless both the *uri* and *data* member of the SpeakData structure can be NULL. This approach makes it possible to combine the old TtsSource source callback function with the new TtsProcessEx function.

When using the source call-back method, the input text must be encoded with the native character set for the active language. Other character sets are only supported if the input is specified via the SpeakData structure. Refer to the description of the SpeakData structure for an overview of all character sets or the "RealSpeak Languages" appendix for a list of the native character sets.

TTSDESTCB

```
Typedef void* (*TTSDESTCB)(
                void* pAppData,
                U16 nDataType,
                void* pData,
                U32 nDataSize,
                U32* pnBufferSize);
```

Purpose: This callback is invoked when the TTS engine instance needs to deliver output data to the application.

The main input parameters of the callback are the address of an application-provided buffer containing output data and the size in bytes of the data. The application provides the buffer for the engine to fill using the return value of the function, which is a pointer to the next buffer to be filled. The size of this application buffer is set in the *pnBufferSize parameter before the function returns. The first call to TTSDESTCB passes in NULL for the output buffer and 0 for the data size, indicating that the TTS engine instance has not yet been given a buffer to fill. This also occurs each time the engine has finished processing a message unit.

In the simplest case, the application allocates a single buffer and returns its address every time, but the application might have a queue of buffers to prevent unnecessary copying of data.

Parameters:

pAppData	Application data pointer that was passed into TtsInitializeEx or TtsInitialize
nDataType	Data type that is being delivered. Currently only TTS_OUTPUTTYPE_PCM is supported
pData	Pointer to the output data buffer
nDataSize	Size of the buffer in bytes. For optimal performance in client-server mode, the buffer size should be set to 4k (4096) bytes.
pnBufferSize	[out] Size in bytes of the “new” data buffer passed back via the return value

Return Value(s):

Pointer to the next output buffer

TTSEVENTCB

```
Typedef TTSRETVAL (*TTSEVENTCB)(
                    void* pAppData,
                    Void* pBuffer,
                    U16 nBufferSize,
                    U16 nEvent);
```

Purpose: This callback is used to return markers to the application. Each marker represents a single event. There's one call to TTSEVENTCB for each separate marker. A marker is thrown before the call to TTSDESTCB that delivers the first audio sample aligned with it. In other words, the user receives the marker info in advance of the corresponding speech.

The different event types are explained under the TTS_Event data type topic.
The user has to specify which marker types he wants to receive. He can do that by calling TtsSetParam() or TtsSetParams() for the TTS_MARKER_MODE_PARAM parameter. See the description of the TTS_Marker data structure for a description of the supported marker types.
Note: The SENTENCEMARK marker is always generated.

Parameters:

pAppData — Application data pointer that was passed into TtsInitialize

pBuffer — Pointer to a buffer containing an event type specific structure providing the event related information; the type of the structure for each event type is listed below.

Event type	Structure type
TTS_EVENT_BOOKMARK	TTS_BookMark
TTS_EVENT_PARAGRAPHMARK	TTS_ParagraphMark
TTS_EVENT_SENTENCEMARK	TTS_SentenceMark
TTS_EVENT_WORDMARK	TTS_WordMark
TTS_EVENT_PHONEMEMARK	TTS_PhonemeMark

nBufferSize — Size of the buffer in bytes

nEvent — Type of event that occurred. Can be one of the following: TTS_EVENT_SENTENCEMARK, TTS_EVENT_BOOKMARK, TTS_EVENT_WORDMARK, TTS_EVENT_PHONEMEMARK, TTS_EVENT_PARAGRAPHMARK

Error Codes

This is a list of all the possible error codes returned by RealSpeak SDK methods:

TTS_SUCCESS,
TTS_ERROR,
TTS_E_WRONG_STATE,
TTS_E_SYSTEMERROR,
TTS_E_INVALIDINST,
TTS_E_BADCOMMAND,
TTS_E_PARAMERROR,
TTS_E_OUTOFMEMORY,
TTS_E_INVALIDPARM,
TTS_E_MISSING_SL,
TTS_E_MISSING_FUNC,
TTS_E_BAD_LANG,
TTS_E_BAD_TYPE,
TTS_E_BAD_OUTPUT,
TTS_E_BAD_FREQ,
TTS_E_BAD_VOICE,
TTS_E_NO_MORE_MEMBERS,
TTS_E_NO_KEY,
TTS_E_KEY_EXISTS,
TTS_E_BAD_HANDLE,
TTS_E_TRANS_EMAIL,
TTS_E_NULL_STRING,
TTS_E_INTERNAL_ERROR,
TTS_E_NO_MATCH_FOUND,
TTS_E_NULL_POINTER,
TTS_E_BUF_TOO_SMALL,

TTS_W_UDCT_ALREADYLOADED,
TTS_E_UDCT_INVALIDHNDL,
TTS_E_UDCT_NOENTRY,
TTS_E_UDCT_MEMALLOC,
TTS_E_UDCT_DATAFAILURE,
TTS_E_UDCT_FILEIO,
TTS_E_UDCT_INVALIDFILE,
TTS_E_UDCT_MAXENTRIES,
TTS_E_UDCT_MAXSOURCESPACE,
TTS_E_UDCT_MAXDESTSPACE,
TTS_E_UDCT_DUPLSOURCEWORD,
TTS_E_UDCT_INVALIDENGHNDL,
TTS_E_UDCT_MAXENG,
TTS_E_UDCT_FULLENG,
TTS_E_UDCT_ALREADYINENG,
TTS_E_UDCT_OTHERUSER,
TTS_E_UDCT_INVALIDOPER,
TTS_E_UDCT_NOTLOADED,
TTS_E_UDCT_STILLINUSE,
TTS_E_UDCT_NOT_LOCAL,

```
TTS_E_UDCT_COULDNOTOPENFILE,
TTS_E_UDCT_FILEREADERROR,
TTS_E_UDCT_FILEWRITEERROR,
TTS_E_UDCT_WRONGTXTDCTFORMAT,
TTS_E_UDCT_LANGUAGECONFLICT,
TTS_E_UDCT_INVALIDENTRYDATA,
TTS_E_UDCT_READONLY,
TTS_E_UDCT_ACTIONNOTALLOWED,
TTS_E_UDCT_BUSY,
TTS_E_UDCT_PRIORITYINUSE,
TTS_E_UDCT_ALREADYENABLED

TTS_E_MODULE_NOT_FOUND,
TTS_E_CONVERSION_FAILED,
TTS_E_OUT_OF_RANGE,
TTS_E_END_OF_INPUT,
TTS_E_NOT_COMPATIBLE,
TTS_E_INVALID_POINTER,
TTS_E_FEAT_EXTRACT,
TTS_E_MAX_CHANNELS,
TTS_E_ALREADY_DEFINED,
TTS_E_NOT_FOUND,
TTS_E_NO_INPUT_TEXT,

/* Client/Server errors */
TTS_E_NETWORK_PROBLEM,
TTS_E_NETWORK_TIMEOUT,
TTS_E_NETWORK_RETRANSMIT,
TTS_E_NETWORK_FUNCTION_ERROR,
TTS_E_QUEUE_FULL,
TTS_E_QUEUE_EMPTY,

TTS_E_ENGINE_NOT_FOUND,
TTS_E_ENGINE_ALREADY_INITIALIZED,
TTS_E_ENGINE_ALREADY_UNINITIALIZED,
TTS_E_DICTIONARY_ALREADY_UNLOADING,
TTS_E_INSTANCE_BUSY,

TTS_E_NOTINITIALIZED,
TTS_E_NETWORK_CONNECTIONREFUSED,
TTS_E_NETWORK_OPENPORTFAILED,
TTS_E_NETWORK_SENDFAILED,
TTS_E_NETWORK_CONNECTIONCLOSED,
TTS_E_ENGINE_OVERLOAD,

TTS_E_UNKNOWN,

/* License Errors */
TTS_E_LIC_NO_LICENSE,
TTS_E_LIC_LICENSE_ALLOCATED,
TTS_E_LIC_UNSUPPORTED,
TTS_E_LIC_LICENSE_FREED,
TTS_E_LIC_SYSTEM_ERROR,

/* Non Fatal Errors or Warnings */
```

```
TTS_W_WARNING,
TTS_W_ENDOFINPUT,

/* Inet Errors */
TTS_E_INET_FATAL,
TTS_E_INET_INPUTOUTPUT,
TTS_E_INET_PLATFORM,
TTS_E_INET_INVALID_PROP_NAME,
TTS_E_INET_INVALID_PROP_VALUE,
TTS_E_INET_NON_FATAL,
TTS_E_INET_WOULD_BLOCK,
TTS_E_INET_EXCEED_MAXSIZE,
TTS_E_INET_NOT_ENTRY_LOCKED,
TTS_E_INET_NOT_ENTRY_CREATED,
TTS_E_INET_UNSUPPORTED,
TTS_E_INET_UNMAPPED,
TTS_E_INET_FETCH_TIMEOUT,
TTS_E_INET_FETCH_ERROR,
TTS_E_INET_NOT_MODIFIED,
```

RealSpeak Telecom
Software Development Kit

Chapter V

SAPI5 Compliance

RealSpeak
V4.0 Manual

Chapter VIII

SAPI5 Compliance

API Support

This section lists which Microsoft Text-To-Speech API v5.1 functions (Text-To-Speech engine Interface) are supported by ScanSoft. For more details on each of these functions, see the chapters on Text-To-Speech Engine Interface in the Microsoft Speech SDK v5.1 Reference. For more information, see the language specific manuals.

Text-To-Speech engine Interface

Interface	Function Name	Availability
ISpTTSEngine	Speak	Supported
	GetOutputFormat	Supported
ISpTTSEngineSite	ISpEventSink	Supported
	GetActions	Supported
	Write	Supported
	GetRate	Supported
	GetVolume	Supported
	GetSkipInfo	Supported
	CompleteSkip	Supported

Chapter VIII

Text-To-Speech Interface

With the exception of IsUISupported and DisplayUI, the Microsoft SAPI5 layer supports all functions of the ScanSoft Text-To-Speech engine interface.

Interface	Function Name	Availability
IspVoice	SetOutput	Supported
	GetOutputObjectToken	Supported
	GetOutputStream	Supported
	Pause	Supported
	Resume	Supported
	SetVoice	Supported
	GetVoice	Supported
	Speak	Supported
	SpeakStream	Supported
	GetStatus	Supported
	Skip	Supported
	SetPriority	Supported
	GetPriority	Supported
	SetAlertBoundary	Supported
	GetAlertBoundary	Supported
	SetRate	Supported
	GetRate	Supported
	SetVolume	Supported
	GetVolume	Supported
	WaitUntilDone	Supported
	SetSyncSpeakTimeout	Supported
	GetSyncSpeakTimeout	Supported
	SpeakCompleteEvent	Supported
	IsUISupported	Not Supported
	DisplayUI	Not Supported

Chapter VIII

SAPI5 Interface

In this section you will find an alphabetical list of member functions of the SAPI5 Text-To-Speech interface (IspVoice).
For a description of each member function, see the chapter on Text-To-Speech Interfaces (ISpVoice), in the Microsoft Speech SDK v5.1 Reference.

ISpVoice Interface

This interface is the only interface for the application to access the Text-To-Speech engine. The ISpVoice interface enables an application to perform text synthesis operations. Applications can speak text strings and text files, or play audio files through this interface. All of these can be done synchronously or asynchronously.
Applications can choose a specific TTS voice using ISpVoice::SetVoice. The state of the voice (for example, rate, pitch, and volume), can be modified using SAPI XML tags that are embedded into the spoken text. Some attributes, like rate and volume, can be changed in real time using ISpVoice::SetRate and ISpVoice::SetVolume. Voices can be set to different priorities using ISpVoice:: SetPriority.
ISpVoice inherits from the ISpEventSource interface. An ISpVoice object forwards events back to the application when the corresponding audio data has been rendered to the output device.

ISpVoice::ISpEventSource

No engine specific remarks.

ISpVoice::SetOutput

The ScanSoft Text-To-Speech engine supports only 8 kHz in this product. If the application chooses other frequencies, then the Microsoft SAPI5 layer will use conversion software installed in the PC, which might cause speech quality degradation.

ISpVoice::GetOutputObjectToken

See ISpVoice::SetOutput.

ISpVoice::GetOutputStream

No engine specific remarks.

ISpVoice::Pause

No engine specific remarks.

ISpVoice::Resume

No engine specific remarks.

ISpVoice::SetVoice

No engine specific remarks.

ISpVoice::GetVoice

No engine specific remarks.

ISpVoice::Speak

No engine specific remarks.

ISpVoice::SpeakStream

No engine specific remarks.

ISpVoice::GetStatus

No engine specific remarks.

ISpVoice::Skip

No engine specific remarks.

ISpVoice::SetPriority

No engine specific remarks.

ISpVoice::GetPriority

No engine specific remarks.

ISpVoice::SetAlertBoundary

No engine specific remarks.

ISpVoice::GetAlertBoundary

No engine specific remarks.

ISpVoice::SetRate

No engine specific remarks.

ISpVoice::GetRate

No engine specific remarks.

ISpVoice::SetVolume

The default volume of ScanSoft voices is 90 instead of 100.

ISpVoice::GetVolume

The default volume of ScanSoft voices is 90 instead of 100.

ISpVoice::WaitUntilDone

No engine specific remarks.

ISpVoice::SetSyncSpeakTimeout

No engine specific remarks.

ISpVoice::GetSyncSpeakTimeout

No engine specific remarks.

ISpVoice::SpeakCompleteEvent

No engine specific remarks.

ISpVoice::IsUISupported

This member function is not supported by ScanSoft's Text-To-Speech engine.

ISpVoice::DisplayUI

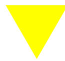
This member function is not supported by ScanSoft's Text-To-Speech engine.

Chapter VIII

SAPI5 XML Tags

In this section you will find an alphabetical list of the Text-To-Speech XML tags that are supported by Microsoft SAPI5. XML tags can be embedded in the input text to change the Text-To-Speech output. For each XML tag, you will find the following information:

Description	Gives a description of the XML tag
Syntax	Displays the syntax of the XML tag
Comments	Gives remarks that are specific to ScanSoft's support of the XML tag
Example	Shows how to use the XML tag



NOTE

12. Only correctly specified XML tags are converted to internally embedded commands. Incorrectly specified control tags are treated as white spaces.

13. The Text-To-Speech control tags that are not supported are not described.

Please see the "Microsoft Speech SDK, V5.1" reference, chapter "Text-To-Speech Interface", for more details on the use and syntax of XML tags, as well as on each XML tag separately.

Control tag	Availability
<Bookmark>	Supported
<Context>	Supported
<Emph>	Supported
<Lang>	Supported
<Partofsp>	Supported
<Pitch>	Not supported
<Pron>	Supported
<Rate>	Supported
<Silence>	Supported
<Spell>	Supported
<Voice>	Supported
<Volume>	Supported

Chapter VIII

Bookmark

Description

This XML tag indicates a bookmark in the text.

Syntax

<bookmark mark=string/>

Comments

None.

Example

This sentence contains a
<bookmark mark=”bookmark_one”/> bookmark.

For more detailed information, see the “Microsoft Speech SDK V5.1” reference, chapter “Text-To-Speech Interface” and “XML TTS Tutorial”.

Context

Description

This XML tag sets the context for the text that follows, determining how specific strings should be spoken.

Syntax

<Context ID=string> Input Text </Context>

Comments

1. The following context types are supported:

   \context ID="date_mdy"\
   \context ID="date_dmy"\
   \context ID="date_ymd"\
   \context ID="date_ym"\
   \context ID="date_my"\
   \context ID="date_dm"\
   \context ID="date_md"\
   \context ID="date_year"\
   \context ID="time_timeofday"\
   \context ID="time_hms"\
   \context ID="time_hm"\
   \context ID="time_ms"\
   \context ID="number_decimal"\
   \context ID="currency"\

2. Some languages do not support this XML tag. See the release note for language specific limitations.

Example

Today is <context ID="date_mdy">12/22/99</Context>.

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-To-Speech Interface" and "XML TTS Tutorial".

Chapter VIII

Emph

Description

This XML tag emphasizes the next sentence to be spoken.

Syntax

<Emph> Input text </Emph>

Comments

This tag is only supported by the ScanSoft engine to emphasize the whole sentence.

Example

<emph>John and Peter are coming tomorrow</emph>.

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-To-Speech Interface" and "XML TTS Tutorial".

Lang

Description

This XML tag indicates a language change in the text. This tag is handled by the Microsoft SAPI5 Layer.

Syntax

<Lang langid=string> Input text </Lang>

Comments

None.

Example

<lang langid="409"> A U.S. English voice should speak this sentence. </lang>

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-To-Speech Interface" and "XML TTS Tutorial".

Partofsp

Description

This XML tag indicates the part-of-speech of the next word. This tag is effective only when the word is in the Lexicon and has the same part-of-speech setting as in the Lexicon.

Syntax

<Partofsp Part=string> word </Partofsp>

Comments

The following part-of-speech types are supported:

- <Partofsp Part=”noun”>
- <Partofsp Part=“verb”>
- <Partofsp Part=“modifier”>
- <Partofsp Part=“function”>
- <Partofsp Part=“interjection”>
- <Partofsp Part=“unknow”>

Example
<Partofsp Part="noun"> A </Partofsp> is the first letter of the alphabet.

For more detailed information, see the “Microsoft Speech SDK V5.1” reference, chapter “Text-To-Speech Interface” and “XML TTS Tutorial”.

Pitch

Description

This XML tag is used to control the pitch of a voice.

Syntax

<Pitch Absmiddle=string> Input Text </Pitch>

Comments

ScanSoft's RealSpeak Engine does not support this tag.

Example

<pitch absmiddle="5">This is a test.</pitch>

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-To-Speech Interface" and "XML TTS Tutorial".

Pron

Description

The Pron tag inserts a specified pronunciation. The voice will process the sequence of phonemes exactly as they are specified. This tag can be empty, or it can have content. If it does have content, it will be interpreted as providing the pronunciation for the enclosed text. That is, the enclosed text will not be processed as it normally would be.

The Pron tag has one attribute, Sym, whose value is a string of white space separated phonemes.

Syntax

<pron sym=phonetic string> or

<pron sym=phonetic string>Input text</pron>

Comments

The phoneme table can be found in the language specific manual: "ScanSoft Telecom RealSpeak SAPI5 V3.51, User's Guide for <language>".
If no phoneme table is available for a specific language, then this tag is not supported for that language.

Example

<pron sym="h eh 1 l ow & w er 1 l d"> hello world </pron>

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-To-Speech Interface" and "XML TTS Tutorial".

Rate

Description

The Rate tag controls the rate of a voice. The tag can be empty, in which case it applies to all subsequent text, or it can have content, in which case it only applies to that content.
The Rate tag has two attributes, Speed and AbsSpeed, one of which must be present. The value of both of these attributes should be an integer between negative ten and ten. Values outside this range may be truncated by the engine (but are not truncated by SAPI). The AbsSpeed attribute controls the absolute rate of the voice, so a value of ten always corresponds to a value of ten, a value of five always corresponds to a value of five.

Syntax

<rate absspeed=number>Input text</rate>

or

<rate speed=number>Input text</rate>

Comments

None.

Example

<rate absspeed=”5”>This is a sentence.</rate>

or

<rate speed=”5”>This is a faster sentence. </rate>

<rate speed=”-5”>This is a slower sentence. </rate>

For more detailed information, see the “Microsoft Speech SDK V5.1” reference, chapter “Text-To-Speech Interface” and “XML TTS Tutorial”.

Chapter VIII

Silence

Description

The Silence tag inserts a specified number of milliseconds of silence into the output audio stream. This tag must be empty, and must have one attribute, Msec.

Syntax

<silence msec=number>Input text

Comments

None.

Example

<silence msec=”500”>This is a sentence.

For more detailed information, see the “Microsoft Speech SDK V5.1” reference, chapter “Text-To-Speech Interface” and “XML TTS Tutorial”.

Chapter VIII

Spell

Description

The Spell tag forces the voice to spell out all text, rather than using its default word and sentence breaking rules, normalization rules, and so forth. All characters should be expanded to corresponding words (including punctuation, numbers, and so forth). The Spell tag cannot be empty.

Syntax

<spell>Input text</spell>

Comments

None.

Example

<spell>UN</spell>

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-To-Speech Interface" and "XML TTS Tutorial".

Chapter VIII

Voice

Description

The Voice tag selects a voice based on its attributes, Age, Gender, Language, Name, Vendor, and VendorPreferred. The tag can be empty, in which case it changes the voice for all subsequent text, or it can have content, in which case it only changes the voice for that content.

The Voice tag has two attributes: Required and Optional. These correspond exactly to the required and optional attributes parameters: ISpObjectTokenCategory EnumerateTokens and SpFindBestToken. The selected voice follows exactly the same rules as the latter of these two functions. That is, all the required attributes are present, and more optional attributes are present than with the other installed voices (if several voices have equal numbers of optional attributes one is selected at random).

For more details, see Object Tokens and Registry Settings in the "Microsoft Speech API V5.1".

In addition, the attributes of the current voice are always added as optional attributes when the Voice tag is used. This means that a voice that is more similar to the current voice will be selected over one that is less similar.
If no voice is found that matches all of the required attributes, no voice change will occur.

Syntax

<voice required=type of info.=info.>Input text</voice>

or

<voice optional=type of info.=info.>Input text</voice>

Comments

None.

Example

<voice required="Gender=Female;Age!=Child">
A female non-child should speak this sentence, if one exists.
</voice> <voice required="Age=Teen">
A teen should speak this sentence - if a female, non-child teen is present, she will be selected over a male teen, for example. </voice>

For more detailed information, see the “Microsoft Speech SDK V5.1” reference, chapter “Text-To-Speech Interface” and “XML TTS Tutorial”.

Volume

Description

The Volume tag controls the volume of a voice. The tag can be empty, in which case it applies to all subsequent text, or it can have content, in which case it only applies to that content.
The Volume tag has one required attribute: Level. The value of this attribute should be an integer between zero and one hundred. Values outside this range will be truncated.

Syntax

<volume level=number>Input text</volume>

Comments

The default volume of ScanSoft voices is 90 instead of 100.

Example

<volume level="50">This is a sentence .</volume>

For more detailed information, see the "Microsoft Speech SDK V5.1" reference, chapter "Text-To-Speech Interface" and "XML TTS Tutorial".

Load ScanSoft User Dictionaries

The user can enable the RealSpeak Solo SAPI5 layer to load the ScanSoft proprietary user dictionary automatically in order to provide better pronunciation for especially proper names, geographical names and so on. This mechanism provides an alternative solution for the SAPI5 lexicon.
The user can specify the list of user dictionaries in the registry per language. What to be registered is as below:

In HKEY_LOCAL_MACHINE\SOFTWARE\ScanSoft\TTS\SAPI5\Install\<Language code (e.g. ENU for American English)>, you need to create strings (sequential number and full pathname of the user dictionary) as below:
"1" "c:\dict1.dbc"
"2" "d:\American English\dict2.bdc"

If there is same entry in multiple dictionaries, then late loaded dictionary will have higher priority. So, the dictionary in "2" will have higher priority than in "1". The full pathname of the user dictionary

should be shorter than 256 characters. The 3 letter language code is as below table:

3-letter language codes

Language name	language code
American English	ENU
Australian English	ENA
Belgian Dutch	DUB
Brazilian Portuguese	PTB
British English	ENG
Canadian French	FRC
Cantonese	CAH
Danish	DAD
Dutch (Netherlands)	DUN
French	FRF
German	GED
Italian	ITI
Japanese	JPJ
Korean	KOK
Mandarin Chinese	MNC
Mexican Spanish	SPM
Norwegian	NON
Polish	PLP
Portuguese (European)	PTP
Russian	RUR
Spanish (European)	SPE
Swedish	SWS

SAPI5 Client/Server

The client/server mechanism for SAPI5 interface provides the user with an environment where multiple thin clients can communicate with one or multiple remote server(s) to request speech output for a given input text.

Required Software

To configure the RealSpeak Telecom SDK in the client/server mode for the SAPI5 interface, you need to install the RealSpeak Telecom SDK package (language independent package) both on the client and on the server side.
Besides the RealSpeak Telecom SDK, you also need to install the Microsoft SAPI5.1 run-time package on the client side.
For the server you need to install at least one language to support basic Text-to-Speech features.

Chapter VIII

The required version of RealSpeak Telecom SDK is 4.0.4 or higher.

Required Hardware

The application developers should decide how many PC's are required for their application. To run the RealSpeak Telecom SDK in the client/server mode by using the SAPI5 interface, you need at least two systems (one for the client and another for the server). Each client PC can connect to up to 64 servers; the number of client systems is not limited.

Installing SAPI5 Layer

To run the SAPI5 interface in the client/server mode, you need a SAPI5 layer module that is different from the one that is used in the standard RealSpeak Telecom SDK. The SAPI5 layer module is called "rs_sapi5_telecom_cs.dll"; it has to be copied to the directory "<install path (e.g. "c:\program files\scansoft\RealSpeak v4.0")>\speech\components\common".

Change in the Registry

To enable the SAPI5 in client/server mode, you need to modify the registry. In the registry, for the key <\HKEY_CLASSES_ROOT\CLSID\{E1E6344F-DDF6-41fb-8F76-FC62CDFC3FF5}\InprocServer32>, you need to change the default value from "rs_sapi5_telecom.dll" to "rs_sapi5_telecom_cs.dll".
If you want to set it back to in-process mode, you need to change "rs_sapi5_telecom_cs.dll" to "rs_sapi5_telecom.dll".

Modifications in the Configuration File

You must add the server information to the configuration file in the client system. The configuration file (ttsserver.xml) can be found in the directory "<install path (e.g. "c:\program files\scansoft\RealSpeak v4.0")>\config".

Chapter VIII

Add Server Information

The list of servers can be added to “ttsserver.xml” as indicated below:

```
<tts_servers>
  <tts_server name="10.151.11.31">
   All
  </tts_server>
  <tts_server name="10.151.11.32">
   American English,Australian English
  </tts_server>
</tts_servers>
```

You can specify multiple servers up to a maximum of 64 servers; you can also specify supported languages for each server. If the server supports all languages, then you can specify “All”.
If the server supports some languages, then you can specify the list of languages and “,” is used as a delimiter between languages. An example is “German, French”.

Changes in the Registry

To allow the SAPI5 layer to read the server information, you need to specify the location of SAPI5 configuration file in the registry.

In <\HKEY_LOCAL_MACHINE\SOFTWARE\ScanSoft\RealSpeak 4.0>, you need to specify the SAPI5 configuration file as in the example below:

SAPI5ConfigFiles=c:\Program Files\ScanSoft\RealSpeak 4.0\config\ttsserver.xml

Load User Dictionaries

To load the user dictionaries in the client/server mode, you need to perform two tasks. The first task is to modify the configuration file in the server to specify a directory that has user dictionaries.
In the configuration file you can specify a directory like below example:

```
<dictionary_default_path>c:\usrdict</dictionary_default_path>
```

In the client you can set a list of user dictionaries by listing them in the registry. You can find the information in chapter 5 in this manual.

Chapter VIII

The SAPI5 layer in the client checks the registry to find a list of user dictionaries that are to be loaded. It sends the list of user dictionaries to the server that loads the dictionaries from the directory specified in <dictionary_default_path>.
All user dictionaries that are to be loaded must exist in the directory specified in <dictionary_default_path>.
Support is possible only for ScanSoft proprietary user dictionary; the format of the user dictionary must be binary. The ScanSoft proprietary user dictionary can be edited by UDE (User Dictionary Editor).

Microsoft Lexicon

The SAPI5 lexicons in the client are not supported in this version.

Enable Logging

To enable the logging facility in the server, please read the "Configuration Files" section of the "User Configuration" chapter. This section provides the information on how to enable the server logging , how to set the logging level and so on, by modifying the server configuration file (by default, ttsserver.xml).

The client requires another action than the server. Do not modify the configuration file; instead, set a number of values in the registry so as to enable the logging mechanism or to set the logging level.
In
<\HKEY_LOCAL_MACHINE\SOFTWARE\ScanSoft\RealSpeak 4.0\Log>,
you can specify the value in the registry as mentioned below:

Enable=<TRUE or FALSE>
Log File=File name (e.g. c:\log.txt)
Log Level=(0 or 1)
Log Size=size (in byte) of log file (e.g. 100000000)

The higher the log level, the more information will be logged in the logging file.
In
<\HKEY_LOCAL_MACHINE\SOFTWARE\ScanSoft\RealSpeak 4.0\Log>
the key "Log" is not automatically created by the RealSpeak Telecom SDK installation package. So you need to create the key "Log" manually.

Chapter VIII

RealSpeak Telecom
Software Development Kit

Chapter VI

SSML Support

RealSpeak
V4.0 Manual

Chapter IX

SSML Support

Introduction and Purpose

SSML (Speech Synthesizer Markup Language) is part of a set of markup specifications by the W3C for voice browsers. SSML was designed to provide a rich, XML-based markup language for assisting the generation of synthetic speech in web and other applications. The essential role of the markup language is to provide a standard way to control aspects of speech such as pronunciation, volume, pitch and rate.
The Telecom RealSpeak/Host SDK provides a built-in preprocessor that supports a large portion of the SSML 1.0 September 2004 Recommendation (REC). Moreover RealSpeak extends SSML with a number of Scansoft specific elements/attributes.
The set supported by Scansoft is called "ScanSoft SSML" (4SML).
Please refer to language specific documentation for language-specific support for certain tags.

Links

Some links to related W3C specifications:

- http://www.w3.org/TR/2002/WD-speech-synthesis-20020405 "Speech Synthesis Markup Language Specification –W3C Working Draft 5 April 2002"
- http://www.w3.org/TR/2002/WD-speech-synthesis-20021202 "Speech Synthesis Markup Language Specification 1.0 – W3C Working Draft 02 December 2002"
- http://www.w3.org/TR/2004/REC-speech-synthesis-20040907 "Speech Synthesis Markup Language Specification Version 1.0 – W3C Recommendation 7 September 2004"

SSML Compliance

Support for the SSML 1.0 REC September 2004

As mentioned before, we aim to be compliant with the W3C specification. At the time of writing, this was the September 2004 Recommendation. We support all elements/attributes specified in the specification, regardless of their rating ("MUST", "REQUIRED", "SHALL", "SHOULD", "RECOMMENDED", "MAY" "OPTIONAL") except where this proves hard to implement due to the nature of the RealSpeak engine. If so, this markup element will be

detected, parsed, but ignored. This applies to the following elements and or properties:

1. The "IPA" alphabet is not supported for <phoneme> elements (SSML "should" level conformance requirement). Phoneme strings must be transcribed using the Scansoft proprietary phonemic alphabet, being "UNIPA" (which we believe to be a more 'developer-friendly' solution). **Error! Unknown document property name.** will fall-back to the element's content when the specified alphabet is not "unipa". Note that the requirement that vendor-defined alphabets must be of the form "**x-organization**" or "**x-organization-alphabet**" is not yet adhered to.
2. <emphasis>: "none" level is not supported. Using this element does not necessarily lead to audible differences as the system may elect ignoring these targets for realizing optimal natural speech output.
3. <mark> only supports marks with names that are unsigned 32-bit integers. Marks that do not meet this requirement are ignored.
4. <voice>
    a. The "age", "gender", "name" and "variant" attribute are only supported when specified together with "xml:lang".
    b. The "age" attribute is supported. But to make this attribute useful, a set of voices with varying age over the same language and gender needs to be installed. At the time of writing this would require the use of custom voices.
5. <prosody>
    a. Duration, pitch, pitch-range, and contour values are ignored.
    b. The volume is not persistent over voice switches. (Rate is neither but that's conformant with the spec.)
6. <break strength="none"> will only have will only have an audible effect when without the <break> element, the TTS engine would have inserted a sentence break.
7. <meta>: http-equiv is not supported
8. <sayas>
   Standardized values for the <sayas> attributes will be published in a W3C Group Note. For an extensive list of RealSpeak 4SML supported say-as attributes and attribute values, see the "Say-as Support" section further in this document. Please note that Scansoft is involved in this Working Group and is committed to support any guidelines that may follow from this.

Legacy support for the SSML 1.0 WD December 2002

RealSpeak supports the elements/attributes of the December 2002 SSML 1.0 WD with the following exceptions:

1. The same limitations as listed under the September 2004 REC compliance section apply except for the <break> element (see below)
2. <break>
    a) Symbolic values for the “time” attribute are ignored. They are supported when specified via the April 2002 WD “size” attribute.
    b) The value “none” for the “time” attribute is processed in the same way as the value “none” for the “strength” attribute (available under the 1.0 REC). It has the effect that a normally inserted sentence break will be removed. . Whereas according to the December 2002 spec the “time” value "none" indicates that a normal break boundary should be used.
3. <prosody rate> is always interpreted as per the final SSML 1.0 REC. April 2002 and December 2002 spec’s interpret a bare number as a word-per-minute value, while in the final SSML recommendation a bare number is a multiplier of the default rate. It proved not feasible to auto-detect which specification the user is working against (the <speak> element will specify the version as “1.0” for all these versions).

Legacy Support for the SSML 1.0 WD April 2002

RealSpeak supports the elements/attributes of the April 2002 SSML 1.0 WD with the following exceptions:

1. The same limitations as listed under the September 2004 REC compliance section apply except for the <break> element (see below)
2. <break>
    a) We continue to support the time attribute with symbolic values. However, avoid using it in new designs as it is considered ‘deprecated’.
    b) The value “none” for the “size” attribute is processed in the same way as the value “none” for the “strength” attribute (available under the 1.0 REC). It has the effect that a normally inserted sentence break will be removed. . Whereas according to the April 2002 spec the “size” value "none" indicates that a normal break boundary should be used.
3. Rate is always interpreted as per the final SSML 1.0 REC. See the previous section on the Dec 2002 WD for more details.
4. Support for the <say-as> element is language specific. Note that RealSpeak supports both the <sayas> syntax of

the April 2002 SSML which specifies the “type” attribute and the Dec 2002 and more recent versions which specify the “interpret-as”, “format” and “detail” attributes.

Volume Scale Conversion

The realization of volume numerical values is SSML conformant. The default value is “100”. and the scale is amplitude linear. Note that although the SSML specifies that the range is 0 to 100, we internally support a more extended range (0 to 200). The values above 100 can only be reached via relative changes or the symbolic values “loud” and “x-loud”.
Note that previous RealSpeak versions (v4.0.3 and older) implemented an amplitude logarithmic volume (or dB) scale with “80” as the default value. This default volume maps to the current default value “100”. The old volume scale corresponds with the native volume markup scale listed in the rightmost column of the table below.
The table below describes the mapping between the SSML volume scale and the Scansoft native volume scale (where the volume value is an integer in the range 0 to 100 which can be set via the native <esc>\vol=x\ markup).

SSML value	Amplitude amplification factor	Loudness in dB	SSFT volume value
0	0.00	-∞d B	0
10	0.10	-20.0 dB	13
20	0.20	-14.0 dB	33
30	0.30	-10.5 dB	45
40	0.40	-8.0 dB	53
50	0.50	-6.0 dB	60
60	0.60	-4.4 dB	65
70	0.70	-3.1 dB	70
80	0.80	-1.9 dB	74
90	0.90	-0.9 dB	77
100	1.00	0.0 dB	80
(141)	1.41	+3.0 dB	90
(200)	2.00	+6.0 dB	100

The formula for converting the SSML value V_{ssml} to the amplification factor A is very simple:

$$A = V_{ssml} / 100$$

The formula for converting a non-zero amplification factor A to the corresponding ssft volume value X_{ssft}:

$$X_{ssft} = \mathrm{Round}((20 * \log10(A) / 0.30) + 80)$$

Chapter IX

The formula for converting a non-zero ssft value X_{ssft} to the dB value Y:

$Y (dB) = (X_{ssft} - 80) * 0.3 dB$

The SSML symbolic values are mapped as follows:

SSML value	Symbolic value	Amplitude amplification factor	Loudness in dB
0	silent	0.00	-∞ dB
18	x-soft	0.18	-15 dB
50	soft	0.50	-6 dB
100	medium	1.00	0 dB
(141)	loud	1.41	+3 dB
(200)	x-loud	2.00	+6 dB

Rate Scale Conversion

We fully support SSML rate markup. The following tables/rules can be used to map SSML markup to equivalent SSFT native markup. The default value is “1.00”.

SSML “number” value	Symbolic value	SSML percentage w.r.t. voice default	SSFT native rate value
0.50	x-slow	-50%	1
0.70	slow	-30%	20
1.00	medium	+0%	50
1.60	fast	+60%	70
2.50	x-fast	+150%	100

SSML descriptive and “number” values change the rate against the voice default. All other rate changes are relative against the (XML) parent element.

The formula to convert an SSML <prosody rate=” X_{ssml}”> value into an SSFT <esc>/rate=Y_{ssft}/ value. Do note that rate changes are relative against parent. I.e. you must cumulate your SSML value over all ancestors before converting.

For increasing the rate ($X_{ssml} > 1.0$)

$Y_{ssft} = Round(50 + (X_{ssml} - 1) / 0.03)$

When decreasing the rate ($X_{ssml} < 1.0$)

$Y_{ssft} = Round(50 - (1 - X_{ssml}) * 100)$

Chapter IX

Break Implementation

A <break strength=”xxx”> element is implemented as a pause of a certain duration, so it directly maps to an SSFT <esc>\pause=x\ tag. The only exception is the SSML <break strength=”none”> element which maps to an SSFT <esc>C tag.
The table below specifies the mapping and the corresponding native markup.

Symbolic value	Duration in ms of the pause	Native markup value
x-weak	100	<esc>\pause=100\
weak	200	<esc>\pause=200\
medium	400	<esc>\pause=400\
strong	700	<esc>\pause=700\
x-strong	1200	<esc>\pause=1200\

When using both the ‘time’ and the ‘strength’ attributes, the ‘time’ attribute gets presidency.

Say-as Support

While the W3C SSML 1.0 Recommendation specifies the say-as element and its semantics with the “interpret-as”, “format”, and “detail” attributes, it does **not** define any specific say-as types. Standardized values for the <sayas> attributes will be published in a W3C Group Note. Please note that Scansoft is involved in this Working Group and is committed to support any guidelines that may follow from this. RealSpeak supports both the <sayas> syntax of the April 2002 SSML which specifies the “type” attribute and the Dec 2002 and more recent versions. The RealSpeak supported say-as types following the more recent SSML specifications are listed below. They generally correspond with the types defined by Speechify v3. The table below provides some guidance to their usage. The say-as types for the older versions are listed in the “RealSpeak v4.0 User’s Guide” for each language but use of them is discouraged.

Say-as “interpret-as” attribute	Say-as “format” attribute	Notes and cautions
acronym		Sequences of letters and/or digits are spoken as “words” whenever this is considered natural in the target language. E.g. NATO, UNESCO for English. Else, letters and digits are pronounced individually. E.g. API for English. Use detail "strict" to force spelling mode. In that case punctuation is also spoken (e.g., speaking a comma as “comma”). Acronym with detail “strict” is equivalent to “letters” with detail “strict”.
address		Used for postal addresses.
cardinal		Supported if relevant in the target language. Roman cardinals are often supported.
currency	(peso and dollar for NA Spanish)	Contained text is a currency amount (the currency symbol may be present in the enclosed text). Supports currencies as commonly specified in the country corresponding to the target language. For example, $, £ and ¥.for US English
date	d,dm,dmy,m,md, mdy,my,y,ym,ym d	
decimal		Same as “number” with format “decimal”. Use of the separator for the integral part is optional. E.g. 123456.123 and 123,456.123 are pronounced in the same way for US English.
digits		Numbers must be read digit by digit. Decimal periods or commas should be pronounced as well. Same as “number” with format “digits”.
duration	h,hm,hms,m,ms, s	For example: “duration” with format “hms” is read out as “<h> hour(s), <m> minute(s), and <s> seconds” (assuming xml:lang was specified as “en-US”)
fraction		Same as “number” with format ”fraction”. E.g. to pronounce“1/3” as one third.
letters		Pronounce alphanumerical strings as a sequence of individual letters and/or digits. With detail “strict” punctuation is also spoken (e.g. speaking a comma as

Say-as "interpret-as" attribute	Say-as "format" attribute	Notes and cautions
		"comma"). Letters with detail "strict" is equivalent to "acronym" with detail "strict". For true spelling of all readable characters, use the "interpret-as" value "spell".
measure		A variety of units (e.g., km, hr, dB, lb, MHz) is supported; the units may appear immediately next to a number (e.g., 1cm) or be separated by a space (e.g., 15 ms); for some units the singular/plural distinction may not always be made correctly.
name		Interpret a string as a proper name if possible.
net	email, uri	"email" can be used for email addresses.
number	cardinal, decimal, digits,fraction, ordinal, telephone	Only formats relevant in the target language are supported. All the format values are supported as interpret-as values as well, behaving the same for either syntax.. See the table entries for those aliases for more details.
ordinal		If relevant, see the language-specific User Guide for a list of the supported formats. Same as "number" with format "ordinal".
spell		The characters in the contained text string are pronounced as individual characters.
telephone		Supports telephone numbers as commonly specified in the country corresponding to the target language. See the language-specific User Guide for a list of the supported formats. Use detail="punctuation" to speak punctuation (e.g. speaking a dash as "dash").
time	h,hm,hms	The hour should be less than 24, minutes and seconds less than 60; AM/PM is read out only if explicitly specified. See the language-specific User Guide for a list of the supported formats.
words		This biases the rendering of "word" strings towards speaking them as words instead of pronouncing them as strings of individual letters and digits. However, the characters of a "word" may still be uttered individually for particularly difficult to pronounce character sequences. Meant for acronyms to be read as words.

Chapter IX

SSML Suport

The “say-as” support varies by language. But ScanSoft expects to continue expanding say-as type coverage for languages over time, both by expanding existing say-as types to other RealSpeak languages, and by adding new say-as types. For unsupported say-as types, RealSpeak merely uses its default text normalization rules. This means that even though Japanese doesn’t support say-as type “telephone”, for example, the output almost always sounds correct anyway since RealSpeak is very good at identifying and properly speaking phone numbers in plain text.
The table below lists the supported say-as types for the RealSpeak languages. For the “Say-as Type” column, we use a compressed notation where colons are used to delimit the “interpret-as”, “format”, and “detail” values. Right-most empty values and the colon delimiter are omitted. For example:

acronym = <say-as interpret-as=“acronym”>
currency:dollar = <say-as interpret-as=“currency” format=“dollar”>
acronym::strict = <say-as interpret-as= “acronym” detail=“strict”>

Say-as Type	American/British/Australian/ Indian English, Canadian/Belgian French, French, German, NA Spanish, Spanish	Japanese	All other languages
acronym	yes		
acronym:strict	yes		
address	yes	yes	yes
cardinal	yes	yes	
currency	yes (see language-specific user guide for list of supported currency symbols)		
currency:dollar	yes		
currency:peso	yes (only by NA Spanish[4])		
date	yes		
date:d	yes		
date:day	yes		
date:dm	yes	yes	
date:dmy	yes	yes	
date:m	yes		
date:md	yes	yes	
date:mdy	yes	yes	
date:my	yes	yes	
date:y	yes		
date:ym	yes	yes	
date:ymd	yes	yes	
digits	yes	yes	

Say-as Type	American/British/Australian/ Indian English, Canadian/Belgian French, French, German, NA Spanish, Spanish	Japanese	All other languages
duration	yes		
duration:h	yes		
duration:hm	yes		
duration:hms	yes		
duration:m	yes		
duration:ms	yes		
duration:s	yes		
letters	yes		
letters::strict	yes		
measure	yes		
name	yes		
net:email	yes		
net:uri	yes		
number	yes		
number:cardinal	yes	yes	
number:decimal	yes	yes	
number:digits	yes	yes	
number:fraction	yes	yes	
number:ordinal	yes		
number:telephone	yes		
number:telephone: punctuation	yes		
ordinal	yes		
spell	yes	yes	yes
telephone	yes		
telephone:: punctuation	yes		
time	yes	yes	
time:h	yes		
time:hm	yes	yes	
time:hms	yes	yes	
words	yes		

The Lexicon Element

We support loading of SSFT user dictionaries through the SSML lexicon element. The value for the 'uri' attribute must be a valid URI to an SSFT user dictionary. The dictionary can be in either one of the two supported SSFT formats. Be it textual (typically *.dct or *.tdc) or binary (typically *.dcb *.bdc).

The 'type' attribute is optional. Valid values are:

- "application/edct-bin-dictionary"
- "application/edct-text-dictionary"

When using an HTTP server, remember to add two entries to your MIME table, thus associating the dictionary extension with the correct MIME type. (Other approaches may be possible, depending on the HTTP-server software you're using) For local file access, the following file extensions are correctly mapped, out of the box: ".dct", ".tdc", ".bdc", and ".dcb".
All lexicon elements are parsed, and user dictionaries loaded before starting Text to Speech conversion. Dictionaries are unloaded when the last sample buffer is generated, or when the TTS process is interrupted by some real time event (STOP).
As always, check the SSML documentation for additional detail.

Scansoft SSML Extensions

RealSpeak SSML extensions have an "ssft-" prefix before the name of the element/attribute. The supported extensions are:

- "ssft-dtype" attribute of <speak>, <p> (paragraph) and <s> (sentence) with values "email" or "text" (or "email_"<xxx> and "text_"<xxx> for the Chinese languages, see language specific user's guide). With this attribute, the user can toggle the TTS output behavior between normal text mode and e-mail specific mode. It has the same effect as the native markup "<esc>%x" (with x is "text" or "email").
- "ssft-domaintype" attribute of <speak>, <p> (paragraph) and <s> (sentence) (value e.g. "propernames"). With this attribute ScanSoft vendor-specific domain types (custom G2P modules) can be enabled. It has the same effect as the native markup "<esc>\domain=x\.... <esc>\domain\". To use this tag, a specific custom G2P needs to be installed. See "Custom G2P Dictionaries" chapter of this manual for further explanation.
- The <audio> element supports three extra attributes to control the internet fetching:
    - fetchtimeout: time in to attempt to open and read the audio document. Use the "s" suffix for seconds, "ms" suffix for msec; if there is no suffix, "ms" is assumed. The value must be an unsigned integer.
    - maxage: value for the HTTP 1.1 cache-control max-age directive. This specifies the application is willing to accept a cached copy of the audio document no older than this value. A value of 0 may be used to force re-validating the cached copy with the origin server. In most cases, this attribute should not be present, thus allowing the origin server to control expiration. Use the "s" suffix for seconds, "ms" suffix for msec; if there is no suffix, "s" is assumed. The value must be an unsigned integer.

- maxstale: value for the HTTP 1.1 cache-Control max-stale directive. This specifies the client is willing to accept a cached copy that is expired by up to this value past the expiration time specified by the origin server. In most cases, this property should be set to 0 or not present, thus respecting the expiration time specified by the origin server. Use the "s" suffix for seconds, "ms" suffix for msec; if there is no suffix, "s" is assumed. The value must be an unsigned integer.

API functions

To enable SSML support, set the markup type via TtsSetParam or TtsSetParams API function:

TtsSetParam(hTtsInst, TTS_MARKUP_TYPE_PARAM , MARKUP_4SML);

By default, the markup type is set to “none” which corresponds with support for the native markup format.

TtsSetParam(hTtsInst, TTS_MARKUP_TYPE_PARAM , MARKUP_NONE);

RealSpeak Telecom
Software Development Kit

Chapter VII

Language Identifier 1.0

Programmer's Guide

Chapter VII

Language Identifier 1.0

Language Identifier 1.0: Preface

Overview

The Language Identifier (Language ID) software lets you identify the source language of text strings encoded in the Windows-1252 code page. To synthesize text that may contain instances of multiple languages with a Text-To-Speech (TTS) system, you must first segment the text into the appropriate language and then route it to the appropriate synthesizer. Similarly, you may want a TTS system to handle dynamically generated content. Generally, you do not know the language of dynamically generated content when you compile an application. In this case, you need to identify the language of the text and pass this information to your application which then picks the TTS synthesizer to use.
Accuracy for the Language ID software is nearly 100% for identifying a single language with even a small sample of 50-100 characters. Even heavily intermingled text drawn from dozens of languages can be segmented and identified extremely accurately.

System Requirements

This section covers the requirements for a Language ID software system.

Size requirements

The sire requirements for the Language ID software are as follows:

- Memory: 32 MB RAM minimum, 64 MB recommended

Chapter VII

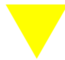
OS requirements

The Language ID software runs on the following operating systems:

- Windows 2000/x86
- Windows NT/x86
- Red Hat Linux AS2.1

Software requirements

The only additional software you need is a C compiler to access the API functions.



NOTES

ScanSoft tests the Language ID software with Microsoft Visual C++ 6.0 on Windows and GCC 3.2.3 on Linux.

Chapter VII

Installing the Language ID software

Installing

The language ID software is an integral part of the RealSpeak Telecom setup. Libraries and include files are installed in the default library and include directories.

Chapter VII

Using the Language ID software

Overview

The Language ID software is able to classify text as being one (or none) of a number of languages.
The application may constrain the identifier to choose from a subset of the installed languages. Call this subset of the installed languages, the active languages.

Language set

The set of languages used in the language identifier must be set at run time from a set of 11 supported languages (enumerated below). The languages selected comprise the set of active languages.
The identifier considers only active languages.

Chapter VII

Available Languages and Codings

For the version 1.0 release, the following set of languages in the Windows-1252 code page is available:

Language	**Language code**
Basque	BAE
Danish	DAD
Dutch	DUX
English	ENX
French	FRX
German	GED
Italian	ITI
Norvegian	NON
Portuguese	PTX
Spanish	SPX
Swedish	SWS

Chapter VII

Language Classification

In its simplest application, language identification takes a sequence of bytes as input and identifies the single language the sequence is most likely to be drawn from.

Tuning Classification

You can tune the Language ID classifications based on preferred languages. The classifier may be configured through the API to prefer active languages. For instance, a French-Canadian installation may prefer English and French to the other available languages. When two languages are given similar scores by the identifier's algorithm, and one is preferred and the other is not, the preferred one is returned. For more information, see "Language Configuration".

Chapter VII

Language ID API Functions

The Language ID software has its own set of API functions. All functions have a name starting with “lid” and can be found in the header file lid.h. Use these functions to control and access the Language ID software.

In this section:

- Data Structures reference
- Function reference

Chapter VII

Data structure reference

The header file lid.h contains two main data structures that are part of the API for the user.

LID_H

This structure describes a handle to a language identifier object.

LID_SCORE_T

This structure describes one active language for the language identification process. Only the active languages will be taken in account for the language identification.

Structure

```
struct LID_SCORE_S {
      char              szLID[4]
      SSFT_U32          value;
};
```

Members

szLID	Language code. This is a 3-letter string that identifies a language. Examples are: “ENX” (English), “GED” (German).

Value	Score value returned by the lid_Idenfity() function call (initially this value has no meaning): the lower this value, the likelier that szLID is the language of the input text. In particular, the probability is proportional with exp(-value/2000)

Chapter VII

lid_ObjOpen()

Syntax:

```
SSFT_ERROR lid_ObjOpen(
                                LID_H * phLid;
};
```

Purpose:

Creates and allocates a new language identifier.

Parameters:

*phLid;	Handle to the new language identifier

Return codes:

SSFT_OK	ok
other code	Error. An overview of all error codes can be found in the header file ssfterror.h.

Chapter VII

lid_ObjClose()

Syntax:

```
SSFT_ERROR lid_ObjClose(
                const LID_H  hLid;
};
```

Purpose:

Removes a language identifier and deallocates all its sources.

Parameters:

hLid;	Handle to language identifier

Return codes:

SSFT_OK	ok
other code	Error. An overview of all error codes can be found in the header file ssfterror.h.

Chapter VII

lid_Identify()

Syntax:

```
void lid_Identify(
                    LID_H  hLid,
                    const char *szText,
                    LID_SCORE_T **ppScore,
                    const SSFT_U16 cScore
};
```

Purpose:

This function identifies the language of the input text (szText) from the set of active languages in the list of language-score pairs (ppScore[0..cScore-1]).

Parameters:

hLid;	Handle to language identifier
szText	The actual input text
ppScore	On input, this list of pointers to language-score pairs defines the languages that need be considered. On return the list is sorted on score from low to high so that the most probable language comes first. see LID_SCORE_T struct for more info [out]
const SSFT_U16	Number of elements in ppScore.

RealSpeak Telecom
Software Development Kit

Chapter VIII

User Configuration

Programmer's Guide

Chapter VI

User Configuration

Overview

This chapter describes the different ways in which a user can tune RealSpeak. It descrribes:

- User Dictionaries
- User Rulesets
- Custom G2P dictionaries
- Custom Voices
- Configuration Files

User Dictionaries

Functional Description

User dictionaries allow the user to specify special pronunciations for specific words or strings of characters (for example, abbreviations). Dictionaries work by substituting a string specified by the user (the “destination string” or “replacement string”) for each occurrence of a word in the original input text that matches the “source string” of a dictionary entry. Source strings cannot contain white space characters, so multi-word entries are not supported.

When a dictionary instance is loaded, the TTS looks up each word in the input text to check if it must be replaced with a destination string from the dictionary.

The case-sensitiveness of the lookup depends on the use of capital letters in the dictionary entry.

a) If the source string does not contain capital letters, the substitution is case-insensitive.
For example when the dictionary contains an entry for the source string "dll", text input keys such as "DLL", "dll", "Dll" and "dLL" will all be substituted.

b) If the source string contains at least one capital letter, the substitution is case-sensitive.
For example when the dictionary contains an entry for "DLL", only the text input key "DLL" will match for that entry.

This is a consequence of the way user dictionaries are used by the TTS engine. The user dictionary is **first** consulted for the original input text key; which means a case-sensitive lookup is performed. If no match is found, the user dictionary is consulted for the lower case version of the input text key.

It is allowed to have two dictionary entries with source strings that only differ in casing. The entry for which the source string contains capital letters takes precedence when the input text key is an exact match.

Suppose the dictionary contains entries for "DLL" and "dll". Then, only the input text key "DLL" will match for entry "DLL". And keys like "dll", "Dll", "dLL" will all match for entry "dll".

The “destination string” of a dictionary entry can be orthographic or phonetic text. Phonetic strings must be presented using the L&H+ phonetic alphabet.

See the language supplement appendix for your specific language for special user dictionary information.

Dictionary substitution rules

- When the same source string occurs more than once in the same subheader, the last occurrence will be chosen to pick the destination string.
- When the same source string occurs in different subheaders with different content type (one phonetic and one orthographic), the occurrence in the first subheader will be chosen to pick the corresponding destination string.
- Only complete words can be matched with; if there’s a source string in the dictionary that is a substring of a word in the input text, there will be no substitution.

Chapter VI

Dictionary Format for RS Host version 4.0

Textual dictionaries must only be encoded in UTF-8 (default) or UTF-16 (auto-detected by the Scansoft User Dictionary Editor tool) and the RealSpeak API. Note that UTF-8 encoded dictionaries may not contain the 3-byte UTF-8 preamble, also known as the UTF-8 BOM or signature.
In general, the dictionary format of textual dictionaries consists of one [Header] label with its properties and several [SubHeader]-[Data] label couples with their properties and data.
Each [SubHeader] describes the expected data properties (such as orthographic or phonetic text) while [Data] describes the actual source string that needs to be replaced by a destination string.

In its most simple form, a dictionary exists of one [Header] label and a [Data] label. Although syntactically correct, there are no actions specified in this dictionary.

The format has been changed compared to version 3.5. The new format can formally be described as:

dictionary format :=

[Header]
{Language = <language code>}

{([SubHeader]
Content=<content type>
Representation=<representation type>
{Language = <language code>}

[Data]
(<source string><separator><destination string><new line>)*
)*}
| {[Data]}

language code := ENA|ENG|ENU|DUN|FRC|GED|…
source string := <word>
destination string := (<word>)*
content type := EDCT_CONTENT_ORTHOGRAPHIC |
EDCT_CONTENT_BROAD_NARROWS
representation type := EDCT_REPR_WSZ_STRING |
EDCT_REPR_SZZ_STRING
separator := tab space
new line := return character (enter)
word := any word

Symbols legend:

symbol	meaning
{…}	optional part; the part between { and } can be occur once but is not required to. example: Language = <language code> does not need to be specified
(…)*	the part between (and) can be occur more then once example: it is possible to have one subheader, multiple subheaders of no subheaders at all.
<…>	the part between < and > specifies a variable string constant example: <language code> can be any of the available language codes (ENU, ENA, FRC…).
a|b	OR part a is specified OR part b is specified. example: as soon as something is specified under [Data], there has to be a [SubHeader] with its properties.

A source string consists of only one word while a destination string can consists of multiple words. Both have to be separated by a tab space. When the destination string consists of phonetic symbols, the string must be preceded by //.

The language string consists of the 3 letter code that identifies a language. Examples of letter codes are ENA, ENG, ENU, FRC, FRF, GED, JPJ, MNC, SPM. A table listing the available language codes can be found in the "RealSpeak Languages" appendix. The language string must be specified; this has to be done or as part of the subheaders or as part of the header (or both). Since TTS doesn't provide support for multiple languages in user dictionaries, the language string has to be the same in header and subheaders.

The content type defines the type of destination string that should be expected; use EDCT_CONTENT_ORTHOGRAPHIC to expect orthographic strings, use EDCT_CONTENT_BROAD_NARROWS to expect phonetic strings.

The content type determines the representation type; when the content type is EDCT_CONTENT_ORTHOGRAPHIC, use EDCT_REPR_WSZ_STRING as representation type. When the content type is EDCT_CONTENT_BROAD_NARROWS, use EDCT_REPR_SZZ_STRING as representation type.

When a dictionary is not built according to these formal rules, the error message TTS_E_UDCT_WRONGTXTDCTFORMAT will be returned when loading the dictionary. When a dictionary is built according to these formal rules it is still possible that the expected result is different or that the error message is returned. It means that the dictionary file has an invalid format.

Possible errors:

- Textual dictionaries must only be encoded in UTF-8 (default) or UTF-16 (auto-detected). Note that all characters in the 7-bit US-ASCII range (hex 20 to7f) are encoded identically whether UTF-8, US-ASCII or for instance Windows-1252 and ISO-8859-1 are used. So dictionaries which only use character codes in the ASCII range can be encoded in for instance Window-1252. If a non US-ASCII character is present (e.g. ä) and the used encoding is for instance Windows-1252, then when the dictionary is loaded via the API an error will be returned and the dictionary will be ignored. Likewise, when the dictionary file is opened in the Scansoft User Dictionary Editor tool (see below), a fatal error will be displayed.
- When as content EDCT_CONTENT_ORTHOGRAPHIC is specified, the destination strings expected for this subheader must consist only of orthographic characters.

When a phonetic string is used, it is interpreted as an orthographic string and no error is returned.

- When EDCT_CONTENT_BROAD_NARROWS is specified as content, the destination strings expected for this subheader must consist only of phonetic characters; an error is returned when a string is found that isn't preceded by //.
- When unknown symbols are used in phonetic content, these are ignored.
- Only one language can be specified; no error is returned but the dictionary is ignored.
- The specified language has to be installed; no error is returned but the dictionary is ignored.

An example dictionary typically looks like this:

```
[Header]
Language = ENU

[SubHeader]
Content=EDCT_CONTENT_ORTHOGRAPHIC
Representation=EDCT_REPR_WSZ_STRING

[Data]
DLL            Dynamic Link Library
Hello  Welcome to the demonstration of the American English Text-to-
Speech system.
info           Information

[SubHeader]
Content = EDCT_CONTENT_BROAD_NARROWS
Representation = EDCT_REPR_SZZ_STRING

[Data]
addr  // '@.dR+Es
```

Dictionary format for older RealSpeak versions (3.x)

If L&H+ phonetic transcriptions are used, they should be preceded by a single forward slash and a plus sign (/+).

Dictionaries can be in text format or binary format. They have the following text format:

- The label [Header] indicates the header section. The content of the header section is optional but the label is not. If the section is left blank, the line after the [Header] label must contain the [Data] label. The header section is made up of a number of fields, whose format is described later in this section. The TTS will store a number of predefined header fields (see the following example). The user can define their own fields, which will be ignored by the TTS. These fields must conform to the format described below or the dictionary will not load.
- The label [Data] indicates the beginning of the data section, which contains the user dictionary entries. The size of the dictionary entries is limited to 40 characters (including null termination) for the source text and to 1024 characters (including null termination) for the destination text.

Header fields have the format:

Field_name = field_text[newline]

The Field_name and field_text must be separated by an equal sign. It does not matter if there are spaces before and after the equal sign.

Dictionary entries have the format:

Target_word[space(s)]replacement_string[newline]

There must be at least one space between the Target_word and the replacement_string. There can be multiple spaces between them.

The following is the equal sample user dictionary as for version 4.0:

```
[Header]
Dictionary Name=us_english_sample.dct
Language=American English
Data Type=ANSI
Date=01/16/2003
[Data]

 DLL  Dynamic Link Library
Hello  Welcome to the demonstration of the American English Text-to-Speech system.
info  Information
addr  /+'@.dR+Es
```

Migration from 3.x to 4.0 format

The following rules have to be kept in mind when converting an old 3.x dictionary to the 4.0 dictionary format:

- Remove [Header] properties Data, Date and Dictionary Name.
- Change the language name to the language code (by example, ‘American English’ to ENU).
- Split the [Data] section in two sections (one for phonetic and one for orthographic destination string).
- Provide a subheader for each category, with the subheader properties (content and representation).
- Replace the /+ sequence by // for all phonetic destination strings.

There are three possible scenarios for conversion:
1. The dictionary only contains orthographic entries
2. The dictionary only contains phonetic entries
3. The dictionary contains both

To demonstrate these scenarios using the rules described above, three examples are shown for each conversion:

Case 1: Orthography only

The new dictionary should look like this:

```
[Header]
Language = ENU

[SubHeader]
Content=EDCT_CONTENT_ORTHOGRAPHIC
Representation=EDCT_REPR_WSZ_STRING

[Data]
DLL  Dynamic Link Library
Hello Welcome to the demonstration of the American English Text-
to-Speech system.
info   Information
```

Case 2: phonetics only

```
 [Header]
Language = ENU

[SubHeader]
Content = EDCT_CONTENT_BROAD_NARROWS
Representation = EDCT_REPR_SZZ_STRING

[Data]
addr   // '@.dR+Es
```

Case 3: both orthography and phonetic

```
[Header]
Language = ENU

[SubHeader]
Content=EDCT_CONTENT_ORTHOGRAPHIC
Representation=EDCT_REPR_WSZ_STRING

[Data]
DLL  Dynamic Link Library
Hello Welcome to the demonstration of the American English Text-
to-Speech system.
info   Information

[SubHeader]
Content = EDCT_CONTENT_BROAD_NARROWS
Representation = EDCT_REPR_SZZ_STRING
```

```
[Data]
addr   // '@.dR+Es
```

User Dictionary API calls

From a developer's point of view, a distinction has to be made between the term 'dictionary' and 'dictionary instance'. A 'dictionary' is the actual file and its content, while a 'dictionary instance' is the loaded version of a dictionary into memory. A 'handle to a dictionary instance' points to a loaded version. A dictionary instance is always linked to one particular TTS engine instance, but one TTS engine instance can be linked to multiple dictionary instances. In the remaining text, the content should make clear whether 'dictionaries' or 'dictionary instances' are being discussed.

For version 4.0, a lot of new dictionary functionality has been created. It is now possible to use more than one dictionary instance simultaneously. Moreover, a priority mechanism is foreseen to determine the order in which dictionaries will be called to perform a lookup.

First, a dictionary instance has to be loaded by calling TtsLoadUsrDictEx. This implicitly also enables a dictionary instance for use, with default priority. All loaded dictionaries thus have the same initial priority. In this case, the order of loading determines the priority

To change priority, a call to TtsEnableUsrDictEx has to be made. Remark that the dictionary instance has to be disabled first by calling TtsDisableUsrDictEx. TtsDisableUsrDictEx can also be used to simply disable (exclude) the dictionary for a lookup. Disabling is not the same as unloading; disabling means that the dictionary instance remains in memory and waits for being enabled again, while unloading a dictionary instance means that the dictionary instance is actually removed from memory.

A typical sequence of dictionary API calls may look like:

```
…
HTTSINSTANCE hInstance;
TtsInitializeEx (&hInstance, pServer, &paramList[0], &instanceData)
…
DictionaryData dictData1;
HTTSDCTEG hDctEg1;
memset(&dictData1,0,sizeof(DictionaryData));
dictData1.uri = "c:\\us_english1.dct";

DictionaryData dictData2;
HTTSDCTEG hDctEg2;
memset(&dictData2,0,sizeof(DictionaryData));
```

```
dictData2.uri = "c:\\us_english2.dct";

TtsLoadUsrDictEx( hInstance,&dictData1,&hDctEg1));
// dictionary 1 is loaded and enabled with
// default priority

TtsProcessEx(hInstance,pSpeakData);
// dictionary 1 is used for lookup

TtsLoadUsrDictEx( hInstance,&dictData1,&hDctEg2));

TtsProcessEx(hInstance,pSpeakData);
// dictionary 2 is looked up, if no entry
// found dictionary 1 is used for lookup

TtsDisableUsrDictEx( hInstance,hDctEg1);
TtsEnableUsrDictEx( hInstance,hDctEg1,0xF));
// priority change of dictionary 1

TtsProcessEx(hInstance,pSpeakData);
// dictionary 1 is looked up, if no entry
// found dictionary 2 is used for lookup

TtsUnloadUsrDictEx( hInstance,hDctEg1);

TtsProcessEx(hInstance,pSpeakData);
// dictionary 2 is used for lookup
TtsUnloadUsrDictEx( hInstance,hDctEg2);

TtsProcessEx(hInstance,pSpeakData);
// no dictionaries used
```

Restrictions on user dictionaries

The following restrictions apply to user dictionaries:

- User dictionary lookup will not be performed when the TTS channel is processing in "word by word" mode (enabled by the escape sequence <ESC>M1)
- You cannot call dictionary functions on a TTS engine instance that is in the state of processing.

Automated User dictionary Loading

User dictionaries can be loaded automatically when a TTS instance is created or the language and/or voice is switched by specifying them in the Server configuration file. Note that this only applies to systems using the Client/Server mode.

See the “Configuration Parameters – default_dictionaries” section in the “Configuration Files” section of the “User Configuration” chapter for more details.

User Dictionary Editor (Windows only)

The Telecom RealSpeak/Host SDK comes with the Scansoft User Dictionary Editor (UDE), which serves as a GUI for creating and editing of user dictionaries. It is installed automatically. Please check out the help documentation that comes with the UDE for detailed instructions. The UDE help file is available via the RealSpeak 4.0 Program group under the Windows Start Menu or directly as “.\speech\components\common\rsude.chm" under the RealSpeak installation directory.

Appendix F

User Rulesets

Introduction

Rulesets allow the user to specify "search-and-replace" rules for certain strings in the TTS "input text". Whereas user dictionaries only support search and replace functionality for literal strings that are complete words, rulesets support any search pattern that can be expressed using regular expressions (e.g. multiple words, part of a word).
The rulesets are applied before any other text normalization is performed, including user dictionary lookup.
The details of how the text normalization can be tuned via user rulesets are described in the next section.
A ruleset is basically a collection of rules; each rule specifies a "search pattern" and the corresponding "replacement spec".
The syntax and semantics of the "search pattern" and the "replacement spec" match those of the regular expression library that is used, being PCRE v5.0 which corresponds with the syntax and semantics of Perl 5. For the Perl 5 regular expression syntax, please refer to the Perl regular expressions main page at http://perldoc.perl.org/perlre.html. For a description of PCRE, a free regular expression library, see http://www.pcre.org/.
More details on the syntax are described in the "Ruleset format" section.
Rulesets can be loaded for a certain TTS instance via the SetParam(s) API function or they can be loaded automatically when a TTS instance is created by specifying them in the Server configuration file.
The rules of a loaded ruleset are applied only when the active language matches the language that is specified in the header section of the ruleset.
Several rulesets can be active simultaneously for the same language; the most recently loaded ruleset is applied first (it has the highest priority).

Tuning of text normalization via rulesets

- The Regular Expression Text-To-Text (RETTT) engine instance applies the rules of the rulesets. It is an optional subcomponent of a Text-To-Text engine instance.
- The rulesets are applied before any other text normalization is performed, including user dictionary lookup. The only transformations on the TTS input text that can occur before the RETTT processing are the transcoding (because the character set does not match the native character set; this is explained in more details in the "Ruleset format" section) and the translation of SSML markup into native RealSpeak markup. So if SSML markup has

been used, it will already be transformed into the native RealSpeak markup format.

- If the TTS input is provided via the user call-back mechanism, it is first collected entirely, before the rules are applied. The first rule of the most recently loaded active ruleset is applied first to the complete input text. Then, regardless of the effect of this rule, the second rule is applied; and so on. The rewriting stops when the last rule of the first loaded ruleset has been applied. In fact it's possible that a later rule changes an input string that was already transformed by a previous rule.

Ruleset format

In general, a ruleset consists of a header section, followed by a data section. The format of a ruleset is described formally below using the same notation as for user dictionaries (see “Symbols Legend” table in the “User Dictionary” section).
A ruleset can be formally described as:
ruleset :=

```
(<comment-line>|<blank-line>)*
<header-section>
<data-section>?
```

Comment lines have the '#' character as the first non-blank character. A blank line is a line consisting entirely of linear whitespace characters. Using regular expression syntax they can be expressed as:

```
comment-line :=
  ^\s*#.*\n
blank-line :=
  ^\s*\n
```

Header Section

The "header" section contains one or more key definitions (the definition of the "language" key is required, see further); each definition can span one line only.

```
header-section :=
    "[header]"\n
    (<comment-line>|<blank-line>|
    <key-definition>)+
```

Comment lines and blank lines can be inserted everywhere.
Key definitions have the following syntax:

```
key-definition :=
  <key-name> = <key-value><comment>?\n
```

Blanks (spaces or tabs) before and after the equal sign are optional. If the key value contains blanks, it must be enclosed in double quotes. If a double quote is needed as part of the value, it needs to be escaped

Appendix F

(\"). The actual syntax of the <key-value> depends on the <key-name>.
A <comment> can follow the <key-value>, it lasts until the end of the line.
comment :=
    #.*$

The only currently supported key names are: "language" and "charset". This means that <key-definition>
can be expressed semantically as:
key-definition :=
  <language-definition>|<charset-definition>

The <language-definition> is required for each header, the value is the 3-letter TTS language code which is also used to specify the language of user dictionaries, see the table in the "RealSpeak Languages" for the list of available language codes.

language-definition :=
  language = <language-code><comment>?\n

language-code := ENA|ENG|ENU|DUN|FRC|GED|…

The <charset-definition> is optional and specifies the character set used for the encoding of the rules. Currently the character set must match the native character set for the language specified in the <language-definition>. See the table in the "RealSpeak Languages" appendix for a list of the native character set for each language.
charset-definition :=
    charset = <charset id> <comment>? \n

charset id :=
    "windows-1252"|"windows-1251"|
    "windows-932"|...

Data Section

The "data" section contains zero or more "rules", a rule can occupy one line only.
data-section :=
    "[data]"\n
    (<comment-line>|<blank-line>|<rule>)*

Comments can also be inserted at the end of a rule and start with a '#' character and span till the end of the line.
A rule has the following syntax:
rule :=
    <search-spec> "-->" <replacement-spec> <comment>? \n

Appendix F

The syntax and semantics of the <search-spec> and the <replacement-spec> matches the one of the used Regular expression library, being PCRE v5.0, this corresponds with the syntax and semantics of Perl 5. For Perl 5 regular expression syntax, please refer to the Perl regular expressions man page at http://perldoc.perl.org/perlre.html. For a description of PCRE, a free regular expression library, see http://www.pcre.org/.
For a detailed description, see the "pcrepattern.html" document in the PCRE distribution package.
If markup is being used (in the source and/or replacement pattern), it must be in the native RealSpeak markup format.
Note that special characters and characters with a special meaning need to be escaped.
Some examples are:

- In the search pattern: non-alphanumerical characters with a special meaning like dot(.), asterisk (*), dollar ($), backslash (\) and so on, need to be preceded with a backslash when used literally in a context where they can have a special meaning (e.g. use * for *). In the replacement spec this applies to characters like dollar ($), backslash (\) and double quote (").
- Control characters like \t (Tab), \n (Newline), \r (Return), etc.
- Character codes: \xhh (hh is the hexadecimal character code, e.g. \x1b for Escape), \ooo (ooo is the octal notation, e.g. \033 for Escape).
- Perl5 also predefines some patterns like “\s” (whitespace) and “\d” (numeric).

For a full description please refer to the Perl5 man pages.

Rule example

/David/ --> "Guru of the month May"
Replaces each occurrence of the string "David" by "Guru of the month May".

Search-spec

In general the format of the search-spec is:
Search-spec :=
<delimiter> <regular-expression> <delimiter> <modifier>*

<delimiter> is usually '/', but can be any non-whitespace character except for digits, back-slash ('\') and '#'... This facilitates the specification of a regular expression that contains '/', because it eliminates the need to escape the '/'.

<modifier> := [imsx]

optional modifiers:

- i (search is case-insensitive);
- m (let '^' and '$' match even at embedded newline characters);
- s (let the '.' pattern match even at embedded newline characters, by default '.' matches any arbitrary character, except for a newline);
- x (allows for regular expression extensions like inserting whitespace and comments in <regular-expression>).

Replacement-spec

The format of the replacement spec is a quoted ("…") string or a non-blank string in case the translation is a single word. It may contain back references of the form $n (n: 1, 2, 3, ...) which refer to the actual match for the n-th capturing subpattern in <search-spec>. E.g. $1 denotes the first submatch. A back reference with a number exceeding the total number of submatches in <search-spec>, is translated into an empty string. A literal dollar sign ($) must be escaped (\$).
Everything following <replacement-spec> and on the same line is considered as comment when starting with '#', else it is just ignored.

Some examples of rules

/<SSFT>/ --> "Scansoft"
Rewrites "<SSFT>" into "Scansoft".
/(Quack)/ --> ($1)
Replaces "Quack" by "(Quack)".
/(Quack)/ --> ($2)
Replaces "Quack" by "()".
/help me/ --> "\x1b\\vol=95\\help me\x1b\\vol=80\\"
Demonstrates the insertion of native markup (volume tag).
Rewrites for instance "Please, help me!" into
"Please, <Esc>\vol=95\help me<Esc>\vol=80\!".
/(\s):-\)(\s)/ --> "$1ha ha$2"
Where "\s" matches any whitespace character,
$1 corresponds with the matched leading whitespace character and $2 corresponds with the matched trailing whitespace character. This rule rewrites for instance " :-) " into " ha ha ".
/(\r?\n)-{3,} *Begin included message *-{3,}(\r?\n)/ --> "$1Start of included message:$2"
Rewrites for instance
---- Begin included message ----
into
Start of included message:

/\x80 ?(\d+)\.(\d{2})\d*/ --> "$1 euro $2 cents"
Rewrites for instance "€9.751" into "9 euro 75 cents".

Appendix F

Restrictions on rulesets

The following restrictions apply to rulesets:

- TTS Markers generated while rulesets are loaded have dummy values (0) for the source position field, because the source positions are only determined after the rulesets have been applied.
- You cannot load or unload rulesets on a TTS engine instance that is in the state of processing.

Effect of rulesets on the TTS performance

The loading of rulesets can effect the performance of the TTS process (Process() and ProcessEx() API function).
An important note is that certain items that may appear in regular expression patterns are more efficient than others.
E.g. a character class (e.g. "[aeiou]") is more efficient than the equivalent set of alternatives (e.g. "(a|e|i|o|u)").
See the "pcreperform.html" main page of the PCRE package for more details.

Ruleset API functions

Rulesets can be loaded and unloaded via the TtsSetParam and TtsSetParams API functions. The TTS_RULESET_LOAD_PARAM parameter allows the loading and enabling of the specified ruleset. Multiple rulesets can be loaded by specifying multiple TTS_RULESET_LOAD_PARAM parameters in one or more TtsSetParam(s) calls.
The most recently loaded ruleset is applied first (so has the highest priority).
If multiple rulesets are specified for one TtsSetParams call then the ones with higher indices in the parameter array argument are loaded last. If an already loaded ruleset is specified, the old copy is first unloaded.
TtsSetParams with TTS_RULESET_UNLOAD_PARAM parameter will unload the specified ruleset.
The information regarding the ruleset is specified via a structure of type TTS_FETCHINFO_T, whose address is stored in the “pObj” field of the parameter value structure (of type TTS_PARAM_VALUE_T).
Note that the TtsSetParams call to "unload" a ruleset should provide the same info as the corresponding "load" call. This is especially needed when a relative URI or path was specified and the SPIINET_URL_BASE
property was set.

Appendix F

See the section “Defined Data Types” in the “RealSpeak API” chapter for a description of the TTS_FETCHINFO_T structure type.

Sample code

A typical sequence of ruleset API calls may look like:

```
…
HTTSINSTANCE hInstance;
TTS_PARAM_T aParamList[1];
TTS_FETCHINFO_T ttsFetchInfo;

TtsInitializeEx (&hInstance, pServer, &Parm, &instanceData)
…

aParamList[0].nParam =
        TTS_RULESET_LOAD_PARAM;
ttsFetchInfo.szUri = "c:\\us_english.trs";
ttsFetchInfo.szContentType =
        "application/x-realspeak-rettt+text";
ttsFetchInfo.hFetchProperties = NULL;
aParamList[0].paramValue.pObj =
        &ttsFetchInfo;

TtsSetParams(hInstance, aParamList, 1);

TtsProcessEx(hInstance,pSpeakData);
/* ruleset “us_english.trs” is applied */

aParamList[0].nParam =
        TTS_RULESET_UNLOAD_PARAM;
aParamList[0].paramValue.pObj =
        &ttsFetchInfo;
TtsSetParams(hInstance, aParamList, 1);
```

Appendix F

Automated ruleset loading

Rulesets can be loaded automatically when a TTS instance is created by specifying them in the Server configuration file. Note that this only applies to systems using the Client/Server mode.
See the "Configuration Parameters – default_rulesets" section in the "Configuration Files" section in the "User Configuration" chapter for more details.

Appendix F

Custom G2P Dictionaries

Nuance's RealSpeak system now offers support for custom G2P dictionaries. A custom G2P dictionary module is an add-on module specifically designed to improve the quality of pronunciation for certain kinds of words (for example, proper names).
Check the language specific manuals for a list of the currently available custom G2P dictionaries. Check with Nuance for the availability of other custom G2P dictionary modules.

One or more custom G2P dictionary modules can be loaded into memory using the API function TtsLoadG2PdictList and unloaded from memory using TtsUnloadG2PdictList.
A custom G2P dictionary module that has been loaded is dynamically enabled/disabled by using the 4SML 'ssft-domaintype' attribute (a proprietary extension to SSML) or the native <esc>\domain\ tag. Refer to the "SSML Support" chapter and the language specific documentation for details.

Appendix F

Custom Voices

ScanSoft's RealSpeak system now offers support for custom voices. Scansoft develops a custom voice at the request of a specific customer, possibly using voice talent contracted by the customer. As part of this process the custom voice font will be given a name that will uniquely identify it for the customer. Each custom voice will go with a specific language (for example, American English).

A custom voice can be selected in the same way as a standard voice, except that when using the TTSPARAM structure, the voice must always be identified via a string, not a number.
RealSpeak allows the selection of a voice in three different ways:

- When the engine instance is initialized. by setting the appropriate values in the TTSPARAM structure, as follows:
  - Set nVoice to TTS_VOICE_USE_STRING
  - Set szVoiceString to string specifying name of the voice to use. The voice name is defined by the customer (see above).

  Example:

  ```
  Parm.nVoice = TTS_VOICE_USE_STRING;
  szVoiceString = "Elizabeth";
  ```

- Using the TtsSetParam(s) function, providing the instance is not busy executing the TtsProcess(Ex) function in another thread.
- Using markup: the SSML <voice> element and voice attribute, the SAPI5 voice tag or the native <esc>\voice\ tag.

Appendix F

Configuration Files

The TTS Server is configured using a XML configuration file, by default "config/ttsserver.xml" within the RealSpeak Telecom installation directory, but this can be changed by specifying one or more -c options (configuration file options) when starting the TTS Server.
Note that when operating RealSpeak in in-process mode, the server configuration file is not used at all (except when using the SAPI5 interface).

If more then one -c option is specified, the configuration parameters in each configuration file override those specified by earlier -c options. This can be used to create OEM or site-specific configuration files that inherit the ScanSoft provided defaults but override a few parameters. This should be done by copying over ttsserver.xml, removing all the parameters except the ones that need to be customized, then customizing those parameter values.

Configuration file format

Within each configuration file, each configuration parameter is specified by one or more XML elements, with the value of the parameter contained within that element.

Here is a sample configuration file followed by a description of the elements and attributes:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl"
 href="ttsserver.xsl"?>

<ttsserver version="4.0.0"
 xmlns="http://www.scansoft.com/rsh40/ttsserver">

  <network_service> </network_service>
  <network_port> 6666 </network_port>
</ttsserver>
```

The sample configuration file consists of the following parts:

- XML declaration:

  ```
  <?xml version="1.0" encoding="ISO-8859-1"?>
  ```

- Style sheet declaration for viewing the file in a Web browser (optional):

Appendix F

```
<?xml-stylesheet type="text/xsl"
 href="ttsserver.xsl"?>
```

- The root element of the document (as specified in the document type declaration), i.e., the container element for parameter elements:

```
<ttsserver version="4.0.0"
 xmlns="http://www.scansoft.com/rsh40/
ttsserver">
```

- One or more elements which are parameters, such as:

```
<network_service> </network_service>
<network_port> 6666 </network_port>
```

Appendix F

Configuration parameters

Not all parameters need to be set; some are optional with the TTS server automatically detecting the appropriate value for that installation.

Single value parameters

The following parameter names are specified as an element name.

Environment Variable Overrides

Element	Description	Default	Optional
<SSFTTTSSDK>	Installation directory	optional parameter, by default auto-detected	yes
<TMPDIR>	Temporary files directory	optional parameter, by default auto-detected	yes
<USER>	User ID	optional parameter, by default auto-detected	yes

Network Parameters

Elements	Description	Default	Optional
<network_service>	TCP/IP service name, if empty network_port is used		
<network_port>	TCP/IP port number	6666	
<network_accept_backlog>	TCP/IP backlog for accepting connections	20	
<network_client_limit>	Maximum number of connections before refusing clients	1000	

Appendix F

Elements	Description	Default	Optional
<network_reuse_addr>	Whether to allow the server to listen for connections on an already active TCP/IP port. By default, false, as doing so exposes a well-known security flaw where other processes could hijack the port afterwards. Only switch the value to true if directed by ScanSoft Technical Support to workaround OS problems with releasing the TCP/IP port on shutdown.	false	
<network_interface>	Network interfaces to listen for connections upon. By default, the server listens for connections on all network interfaces. Uncomment and set this to enhance security in cases where the server should only accept clients from the same host (use a value of 127.0.0.1), or where the server should only accept clients from a single trusted network interface (use the TCP/IP address for that interface). This parameter is optional.	all interfaces	yes
<network_timeout>	Network timeout, in seconds, for the client/server connection. The worst-case for detecting a lost connection is 2 times this value.	60	

Licensing Parameters

Elements	Description	Default
<license_mode>	Licensing mode, default or explicit	default
<license_servers>	See Multiple Value Parameters	27000@localho st

Appendix F

Speak Parameters

Elements	Description	Default
<default_rate>	Default speaking rate on the RealSpeak rate scale of 1 - 100. This value is overridden if the rate is set via the RealSpeak API or markup, and has no effect for SAPI where SAPI always specifies its own default rate.	50
<default_volume>	Default volume level on the RealSpeak volume scale of 0 - 100. This value is overridden if the volume is set via the RealSpeak API or markup, and has no effect for SAPI where SAPI always specifies its own default volume.	80

Miscellaneous Server Parameters

Elements	Description	Default
<default_dictionaries>	See Multiple Value Parameters	
<default_rulesets>	See Multiple Value Parameters	
<dictionary_default_path>	Default path for user dictionaries.	
<run_in_background>	Run in the background versus as an interactive process.	false
<produce_core_files>	Whether to produce core files on crashes for UNIX variants.	true

Internet Fetch Cache Parameters

Elements	Description	Default
<cache_directory>	Directory name for the disk cache. If relative, the file path is relative to the containing configuration file.	${TMPDIR}/ttsserver_cache_${USER}
<cache_total_size>	Maximum size of the disk cache in MB	200
<cache_entry_max_size>	Maximum size of a single disk cache entry in MB	20
<cache_entry_exp_time>	Time when an unused disk cache entry gets purged, in seconds	3600
<cache_low_watermark>	When maximum cache size is reached, what the cache size must be reduced to, in MB	180
<cache_unlock_entries_at_startup>	Reserved for future use, leave unchanged	true

Appendix F

Internet Fetch Parameters

Elements	Description	Default
<inet_proxy_server>	Address a http proxy server to use, e.g. 127.0.0.1	no proxy is used (empty value)
<inet_proxy_server_port>	Port of the http proxy server to use, e.g. 8080, ignored unless inet_proxy_server is non-empty	8080
<inet_user_agent>	User agent name in HTTP/1.1 headers	RealSpeak Host/4.0
<inet_accept_cookies>	Whether to accept HTTP cookies (true or false)	true
<inet_extension_rules>	See Multiple Value Parameters	

Appendix F

Diagnostic and Error Logging Parameters

Elements	Description	Default
<log_file_enabled>	Whether to log errors and diagnostics (true or false)	true
<log_file_base_name>	Error and diagnostic log file base name. This will have "1.xml"and "2.xml" appended for the initial and roll-over log file names. If relative, the file path is relative to the containing configuration file. If empty, messages will go to standard output.	${TMPDIR}/ttsserver_log_${USER}_
<log_file_max_size>	Log file maximum size, in MB	50
<log_level>	Diagnostic log level, by default 0, where 0 enables errors, 1 enables errors and warnings, and higher levels enable diagnostic messages (which may greatly impact performance).	0

Multiple Value parameters

The following parameters can have multiple values, and are specified as a combination of nested elements.

inet_extension_rules

Rules for mapping file name extensions to MIME content types, specified as a sequence of <extension> elements where the name attribute is the extension and the value is the MIME content type.

For example, these are the RealSpeak defaults:

```
<inet_extension_rules>
  <extension name=".alaw"> audio/x-alaw-basic </extension>
  <extension name=".ulaw"> audio/basic </extension>
  <extension name=".wav"> audio/x-wav </extension>
  <extension name=".L16"> audio/L16;rate=8000 </extension>
  <extension name=".txt"> text/plain </extension>
  <extension name=".xml"> text/xml </extension>
  <extension name=".ssml" >application/ssml+xml </extension>
  <extension name=".bdc"> application/edct-bin-dictionary </extension>
  <extension name=".dct"> application/edct-text-dictionary </extension>
  <extension name=".tdc"> application/edct-text-dictionary </extension>
</inet_extension_rules>
```

default_dictionaries

List of default dictionaries to load, where each matching dictionary is loaded when each port is opened. Language and priority attributes are required, and a voice attribute is optional (if not specified, the dictionary is loaded for all voices for that language). The value is the

dictionary path or URI. This is an optional parameter, by default empty.

For example, the following would load american_english.bdc for all American English voices, that dictionary as well as jill.bdc for the American English Jill voice, and no default dictionaries for any other language.

```
<default_dictionaries>
  <dictionary language="American English" priority="1000">
    http://myserver/american_english.bdc
  </dictionary>
  <dictionary language="American English" voice="Jill"
   priority="1001">
    http://myserver/jill.bdc
  </dictionary>
</default_dictionaries>
```

default_rulesets

List of default rulesets to load, where each matching ruleset is loaded when each port is opened. This element is optional; by default no rulesets are loaded. A default ruleset is specified via a <ruleset> element. The "language" attribute is required, the "content-type" and "voice" attributes are optional. The default value for the content-type is "application/x-realspeak-rettt+text".
If the voice attribute is not specified, the ruleset is loaded for all voices for that language. The value is the ruleset path or URI.

For example, the following would load american_english.trs for all American English voices, that ruleset as well as david.trs for the American English David voice, and no default rulesets for any other language.

```
<default_rulesets>
  <ruleset language="American English">
    http:// myserver/american_english.trs
  </ruleset>
  <ruleset language="American English" voice="David">
    http://myserver/david.trs
  </ruleset>
</default_rulesets>
```

license_servers

RealSpeak license server TCP/IP addresses, where at least one license server must be defined, and multiple values are used to configure redundant license servers for fail-over support. See the RealSpeak Licensing Handbook for details on license server configurations and considerations, as well as detailed information on using this parameter properly.

For example, this is the RealSpeak default:

```
<license_servers>
  <server>27000@localhost</server>
</license_servers>
```

RealSpeak Telecom Software Development Kit

Chapter IX

RealSpeak Email Pre-Processor

Programmer's Guide

Appendix L

RealSpeak E-mail Preprocessor

Introduction

The ScanSoft e-mail preprocessor (EMPP) has been developed to analyze a specific type of text: e-mail messages. E-mail messages differ from any average type of text in both structure and contents.

An e-mail message consists of two clearly distinguished parts: the header and the body. A substantial part of the header contains routing and administrative information, which is irrelevant to the user. Both the header and the body contain all kinds of e-mail specific text features, e.g. e-mail addresses, emoticons such as smileys, etc. Furthermore, informal writing is often combined with a lack of grammatical conventions. Spelling rules are frequently violated, punctuation is often omitted, etc.

Although the standard ScanSoft Text-To-Speech system can handle special text items (abbreviations, numbers, dates, etc.), it is not capable of correctly handling all e-mail specific text features. These text features are therefore dealt with by the e-mail preprocessor. The EMPP transforms e-mail specific information into a format that complies with the rules of the standard ScanSoft Text-To-Speech system. The EMPP is a plug-in preprocessing module of the ScanSoft Text-To-Speech system. It replaces the preprocessor of the standard Text-To-Speech system.

In the following sections you will find a description of the functioning of the ScanSoft e-mail preprocessor as well as an overview of its features.

The e-mail preprocessor has two main tasks: processing of the e-mail header and processing of the body of the e-mail message.

The input to the EMPP consists of one or more e-mail messages. In order to process the e-mail header, the EMPP extracts relevant header fields and then provides an intelligent header field reading.

Appendix L

During the processing of the e-mail body, the text is divided into smaller text units, called text-to-speech messages, which are synthesized by the Text-To-Speech system. Text normalization is applied to e-mail specific text features such as e-mail addresses, proper names, emoticons, URLs (Universal Resource Locators), etc. For the text normalization of an e-mail message, the ScanSoft EMPP applies linguistic rules and performs dictionary look-up, in order to yield an adequate phonetic transcription. The EMPP also supports the ScanSoft user dictionary mechanism, which allows the user to customize the output of the e-mail processing.

E-Mail Header Processing

Header Field Extraction

An e-mail message consists of two clearly distinguished parts: the header and the body. The EMPP detects the header and extracts the relevant header fields. Information that is of no interest to the user (such as routing information) is not retained.
The EMPP extracts the following header fields:

From Field	Contains the sender's name and/or address
Date Field	Contains the date and time of sending
Subject Field	Optionally contains the subject of the e-mail

The extraction of the header fields is based on the detection of specific keywords in the e-mail header. The supported keywords for the extraction of the header fields are language specific and are listed in the "E-mail Preprocessor" chapter of the language-specific manuals. Some examples of keywords are listed below. Note that the keywords do not necessarily have to be presented in the same language as the body of the e-mail.

From Field	From: Author: Sender: De: Von:
Date Field	Date: Enviado: Gesendet:
Subject Field:	Subject: Subj: Asunto:

Appendix L

Betreff:

The following is an example of header field extraction. The original header holds information that is irrelevant to the user. After extraction of date, sender and subject, the processed header merely mentions the Date field, the From field and the Subject field:

Original header:

```
From owner-techlink@eva.dc.LSOFT.COM Wed Jan 31 07:04:47 1996
Return-Path: <owner-techlink@eva.dc.LSOFT.COM>
Received: from lhs1.lhs.be by mars.lhs.be (4.1/SMI-4.1)
       id AA03971; Wed, 31 Jan 96 07:04:44 +0100
Received: (from uucp@localhost) by lhs1.lhs.be (8.6.11/8.6.11) id
HAA08429; Wed, 31  Jan 1996 07:02:54 +0100
Received: from smtpgate.cmp.com ([198.80.26.6]) by keystone.cmp.com
with ESMTP
      (1.37.109.14/17.1) id AA029325060; Tue, 30 Jan 1996 18:37:40 –
0500
X-Mailer: Microsoft Mail via PostalUnion/SMTP for Windows NT
Approved-By:  TechWeb <techweb@CMP.COM>
Message-Id:  <1996Jan30.181300.1151.634035@smtpgate.cmp.com>
Date: 30 Jan 96 18:40:28 –0500
Reply-To: TECHLINK-REQUEST@eva.dc.LSOFT.COM
From: TechWeb <techweb@cmp.com>
Organization: CMP Publications, Inc.
Subject:      TechWeb's TechLink newsletter; January 3
To: Multiple recipients of list TECHLINK
            <TECHLINK@eva.dc.LSOFT.COM>
Status: R
```

Extracted header fields:

```
Date:  30 Jan 96 18:40:28 –0500
From: TechWeb <techweb@cmp.com>
Subject:  TechWeb's TechLink newsletter; January 3
```

Header Field Reading

After the header fields have been extracted, they are processed by the EMPP. The header field keywords (see above) are replaced by an introductory message. The remainder of the header fields is processed by the EMPP in order to allow the Text-To-Speech system to intelligently read the fields. See the "E-mail Preprocessor" chapter of the language-specific manuals for the details.

Appendix L

E-Mail body processing

Message Extraction

The e-mail preprocessor splits the body of the e-mail message into text-to-speech messages. This is done on the basis of a number of criteria, such as punctuation, capitalization, layout, intelligent abbreviation handling, etc.
See the "E-mail Preprocessor" chapter of the language-specific manuals for the details.

Text Normalization

An e-mail message typically contains e-mail specific text features, such as e-mail addresses, URLs, file names, emoticons, etc. The EMPP transforms these e-mail specific features into a format that complies with the rules of the standard text normalization of the ScanSoft Text-To-Speech system.
Refer to the "E-mail Preprocessor" chapter of the language-specific manuals for the details.

Customizing the E-Mail Preprocessor

The e-mail preprocessor supports the standard ScanSoft Text-To-Speech SDK user dictionary mechanism, which allows the user to customize the output of the e-mail preprocessor. The user dictionary is consulted both during the header processing and the body processing.
For the details of customization of the reading of the e-mail header and body, refer to the "E-mail Preprocessor" chapter of the language-specific manuals for the details.
For more information on how to build and use user dictionaries, see the **User Dictionaries** section of the "User Configuration" chapter.

Support for markup in E-mail mode

The e-mail preprocessor can, in general, be activated via markup of the input text using the native <ESC>%email tag or via the setting of the "ssft-dtype" (document type) attribute to "email" for the <speak>, <paragraph> or <sentence> element in SSML mode. For Mandarin Chinese and Cantonese the document type should be set to email_xxx where xxx is win950 or win936 (Mandarin only) depending on the desired native character set (see also the language specific user's guides). Note that the e-mail preprocessor can also be activated via the API.

Appendix L

Most tags supported in standard text mode are also supported in e-mail mode apart from the few exceptions listed below.

Native markup

- No support for any of the <ESC>\tn=x\ text normalization (TN) tags except for <ESC>\tn=spell\ and <ESC>\tn=normal\. Markup for the other TN types is simply ignored.
- No support for <ESC>M1 (word-by-word read mode) and <ESC>M3 (line-by-line read mode)

SSMLmarkup

- No support for <say-as> tags except for <say-as interpret-as="spell"> and <say-as type="spell-out">. Markup for the other say-as types is simply ignored.

E-mail Preprocessor API functions

The e-mail preprocessor can be activated via the TtsSetParam and TtsSetParams API functions by setting the TTS_DOCUMENT_TYPE_PARAM parameter to DOC_EMAIL. To disable the E-mail preprocessor the parameter must be set to DOC_NORMAL.

See the "RealSpeak API" chapter for a description of the TtsSetParam(s) functions.

Sample code

A typical sequence of code using the e-mail preprocessor may look like:

```
…
HTTSINSTANCE hInstance;
TTS_PARAM_T aParamList[1];

TtsInitializeEx (&hInstance, pServer, &Parm, &instanceData)
…
/*** Activate the e-mail preprocessor ***/
aParamList[0].nParam =
        TTS_DOCUMENT_TYPE_PARAM;
aParamList[0].paramValue.nNo = DOC_EMAIL;

TtsSetParams(hInstance, aParamList, 1);

/*** Process an e-mail document ***/
```

Appendix L

```
TtsProcessEx(hInstance,pSpeakData);

/*** Deactivate the e-mail preprocessor ***/
aParamList[0].nParam =
        TTS_DOCUMENT_TYPE_PARAM;
aParamList[0].paramValue.nNo = DOC_NORMAL;

TtsSetParams(hInstance, aParamList, 1);
```

RealSpeak Telecom
Software Development Kit

Chapter X

Speechify API

Programmer's Guide

Speechify API

Introduction

RealSpeak Telecom 4.0 and RealSpeak Solo 2.0 support nearly all of the SWItts API of Speechify 3.0 and Speechify Solo 1.0 to ease the migration of existing Speechify based integrations and applications to the next-generation RealSpeak products that incorporate Speechify technology. New software should only be developed using the native RealSpeak APIs or the Microsoft SAPI 5 APIs, however.

For a detailed list of Speechify functionality that is not present in the RealSpeak emulation of the SWItts API, and migration procedures, see the RealSpeak Migration Guide for Speechify Customers.

API Reference

This chapter describes RealSpeak support for the SWItts API, including functions that are not supported. For RealSpeak, the main SWItts API function prototypes, types, error codes, and constants are located in the header file SWItts.h.

Calling convention

The calling convention is dependent on the operating system, and is defined in the SWItts.h header file.
On Windows, all SWItts API functions use the stdcall (or Pascal) calling convention. The header files contain the appropriate compiler directives to ensure correct compilation. When writing callback functions, be sure to use the correct calling convention.

Under Windows:

```
#define SWIAPI __stdcall
```

Under UNIX:

```
#define SWIAPI
```

SDK’s preferred character set

The SDK’s preferred character set varies by language in the same way as the native RealSpeak API. All strings passed to the API by calls to SWIttsSpeak( ), SWIttsSpeakEx( ), and SWIttsDictionaryLoad( ) are converted to the native character set for that language before they are processed internally. Consequently, in RealSpeak, text entered into this function must be representable in the preferred character set even if it is encoded in another character set supported by the API. Bookmark IDs are converted to 0-terminated wide character strings before they are returned to the user.
See the RealSpeak User’s Guide for each language for details.

Result codes

The following result codes are defined in the enum SWIttsResult in *SWItts.h* .

SWItts_ALREADY_EXECUTING_API	This API function cannot be executed because another API function is in progress on this port on another thread.
SWItts_ALREADY_INITIALIZED	SWIttsInit( ) was called when the SWItts API library was already initialized.
SWItts_CONNECT_ERROR	The SWItts API could not connect to the engine.
SWItts_DICTIONARY_ACTIVE	The dictionary is active; it cannot be activated again or cannot be freed until deactivated.
SWItts_DICTIONARY_LOADED	Dictionary is already loaded.
SWItts_DICTIONARY_NOT_LOADED	Dictionary has not been loaded before attempting to activate it.
SWItts_DICTIONARY_PARSE_ERROR	Any error during dictionary parsing.
SWItts_DICTIONARY_PRIORITY_ALREADY_EXISTS	No duplicate priorities are allowed in dictionaries of the same type and language.
SWItts_ERROR_PORT_ALREADY_STOPPING	SWIttsStop( ) was called when the port was already in the process of stopping.
SWItts_ERROR_STOP_NOT_SPEAKING	SWIttsStop( ) was called when the port was not

	speaking.
SWItts_FATAL_EXCEPTION	(Windows only.) A crash occurred within the SWItts API library. This is an unrecoverable error and you should close the application.
SWItts_HOST_NOT_FOUND	Could not resolve the host name or IP address.
SWItts_INVALID_MEDIATYPE	Unsupported MIME content type for the dictionary format or speak text format.
SWItts_INVALID_PARAMETER	One of the parameters passed to the function was invalid.
SWItts_INVALID_PORT	The port handle passed is not a valid port handle.
SWItts_INVALID_PRIORITY	Dictionary priority value is out of range.
SWItts_LICENSE_ALLOCATED	A license has already been allocated for this port.
SWItts_LICENSE_FREED	A license has already been freed for this port.
SWItts_MUST_BE_IDLE	This API function can only be called if the TTS port is idle.
SWItts_NO_LICENSE	There are no purchased licenses available.
SWItts_NO_MEMORY	An attempt to allocate memory failed.
SWItts_NO_MUTEX	An attempt to create a new mutex failed.
SWItts_NO_THREAD	An attempt to create a new thread failed.
SWItts_NOT_EXECUTING_API	An internal error. Notify ScanSoft technical support if you see this result code.
SWItts_PORT_ALREADY_SHUT_DOWN	The port is already closed. You cannot invoke SWIttsClosePort( ) on a port that has been closed.
SWItts_PORT_ALREADY_SHUTTING_DOWN	SWIttsClosePort( ) was called when the port was already being closed.
SWItts_PORT_SHUTTING_DOWN	A command could not be executed because the port is shutting down.
SWItts_PROTOCOL_ERROR	An error in the client/server communication protocol

	occurred.
SWItts_READ_ONLY	SWIttsSetParameter( ) was called with a read-only parameter.
SWItts_SERVER_ERROR	An error occurred on the server.
SWItts_SOCKET_ERROR	A sockets error occurred.
SWItts_SSML_PARSE_ERROR	Could not parse SSML text.
SWItts_SUCCESS	The API function completed successfully.
SWItts_UNINITIALIZED	The SWItts API is not initialized.
SWItts_UNKNOWN_CHARSET	Character set is invalid or unsupported.
SWItts_UNSUPPORTED	Feature is not supported.
SWItts_URI_FETCH_ERROR	Any error during URI access other than SWItts_URI_NOT_FOUND or SWItts_URI_TIMEOUT.
SWItts_URI_NOT_FOUND	URI was not found: the file does not exist or the web server does not have a matching URI.
SWItts_URI_TIMEOUT	Timeout during web server URI access.
SWItts_WINSOCK_FAILED	WinSock initialization failed. (Windows only.)

This is the full set of codes that the API functions return. No functions return all the codes. SWItts_SUCCESS and SWItts_FATAL_EXCEPTION are the only codes that are common to all functions. All functions except SWIttsInit( ) return SWItts_UNINITIALIZED if SWIttsInit( ) was not the first function called.

SWIttsAddDictionaryEntry( )

Mode

Synchronous

Purpose

Adds a list of dictionary entries to the specified dictionary.

```
SWIttsResult SWIAPI SWIttsAddDictionaryEntry (
SWIttsPort                  ttsPortExt,
const char*                 dictionaryType,
const char*                 charset,
SSFT_U32                    numEntries,
SWIttsDictionaryEntry*      entries
);
```

Notes

Currently not supported, always returns SWItts_UNSUPPORTED. Use SWIttsDictionaryLoad( ) instead.

SWIttsCallback( )

Mode

Synchronous. Important: You must not block/wait in this function.

Purpose

User-supplied handler for data returned by the synthesis engine.

```
typedef SWIttsResult (SWIAPI SWIttsCallback) (
SWIttsPort            ttsPort,
SWItts_cbStatus       Status,
void*                 data,
void*                 userData
);
```

Parameters

ttsPort	The port handle returned by SWIttsOpenPortEx( ) or SWITTS_INVALID_PORT (-1) if the callback is called from within SWIttsInit( ), SWIttsOpenPortEx( ), or SWIttsTerm( ).
status	These are enumerated types that are used to inform the callback function of the status of the void *data variable. See table below.
data	Pointer to a structure containing data generated by the engine. This pointer is declared as void * because the exact type varies. The status parameter indicates the exact type to which this pointer should be cast.
userData	This is a void * in which the application programmer may include any information that he wishes to be passed back to the callback function. A typical example is a thread ID that is meaningful to the application. The userData variable is a value you pass to these functions: • SWIttsInit( ) for errors during SWIttsInit( ) • SWIttsTerm( ) for errors during SWIttsTerm( ) • SWIttsOpenPortEx( ) otherwise

This table lists the values of SWItts_cbStatus:

SWitts_cbAudio	Audio data packet. The data structure is a SWIttsAudioPacket shown below.
SWItts_cbBookmark	User-defined bookmark. The data structure is a SWIttsBookMark as shown below.
SWItts_cbDiagnostic	Diagnostic message. The data structure is a SWIttsMessagePacket as shown below. You only receive this message if the SWITTSLOGDIAG environment variable is defined.
SWItts_cbEnd	End of audio packets from the current SWIttsSpeak( ) command. The data is a NULL pointer.

SWItts_cbError	Asynchronous error message. This message is received if an asynchronous API function encounters an error when trying to perform an asynchronous operation such as reading from the network. If you receive this message, consider it fatal for that port. You are free to call SWItts functions from the callback but you should consider the receipt of SWItts_cbError fatal and call SWIttsClosePort( ) to properly clean up the port. This event is always preceeded with a SWItts_cbLogError event that indicates the error reason. The ttsPort argument is SWItts_INVALID_PORT (-1) and the userData argument could be NULL if the failure occurred during SWIttsInit( ), SWIttsOpenPortEx( ), or SWIttsTerm( ). Make sure you check for these possibilities before your code dereferences userData or uses the port number for a lookup.
SWItts_cbLogError	Error message. The data structure is a SWIttsMessagePacket which contains error information, and is described below. The callback may receive the cbLogError and cbDiagnostic events at anytime, whether inside a synchronous or asynchronous function. The user is not allowed to call any SWItts function at this time. If you receive this message, log it to a file, console, etc., and continue execution. The ttsPort argument is SWItts_INVALID_PORT (-1) and the userData argument could be NULL if the failure occurred during SWIttsInit( ), SWIttsOpenPortEx( ), or SWIttsTerm( ). Make sure you check for these possibilities before your code dereferences userData or uses the port number for a lookup.
SWItts_cbPhonememark	Represents information about a phoneme boundary in the input text. The data structure is a SWIttsPhonemeMark shown below.
SWItts_cbPortClosed	The port was successfully closed after a call to SWIttsClosePort( ). The data is a NULL pointer.
SWItts_cbStart	Represents the commencement of audio packets from the current SWIttsSpeak( ) command. The data is a NULL pointer.
SWItts_cbStopped	SWIttsStop( ) has been called and recognized. There is no SWItts_cbEnd notification. The data is a NULL pointer.
SWItts_cbWordmark	Represents information about a word boundary in the input text. The data structure is a SWIttsWordMark shown below

Structures

The **audio packet data structure** is described here:

```
typedef struct {
void *                    samples;
SSFT_U32                  numBytes;
SSFT_U32                  firstSampleNumber;
} SIttsAudioPacket;
```

Structure members:

a)

samples	The buffer of speech samples. You must copy the data out of this buffer before the callback returns as the SWItts API library may free it or overwrite the contents with new samples.
numBytes	The number of bytes in the buffer. This number of bytes may be larger than the number of samples, e.g., if you've chosen a sample format of 16-bit linear, the number of bytes would be twice the number of samples.
firstSampleNumber	The accumulated number of samples in the current SWIttsSpeak( ) call. The first packet has a sample number of zero.

The **message packet data structure** is described here:

```
typedef struct {
time_t                        messageTime;
SSFT_U16                      messageTimeMs;
SSFT_U32                      msgID;
SSFT_U32                      numKeys;
const SSFT_TCHAR**            infoKeys;
const SSFT_TCHAR**            infoValues;
const SSFT_TCHAR*             defaultMessage
} SWIttsMessagePacket;
```

Structure members:

messageTime	The absolute time at which the message was generated.
messageTimeMs	An adjustment to messageTime to allow millisecond accuracy.
msgID	An unique identifier corresponding to one of the SWItts_err[…] defines in SWItts.h. A value of 0 is used for SWItts_cbDiagnostic messages.
numKeys	The number of key/value pairs (the number of entries in the infoKeys and infoValues arrays). For SWItts_cbLogDiagnostic messages, this is always 0. For

	SWItts_cbLogError messages, this may be 0 or greater.
infoKeys/infoValues	Additional information about message, in key/value pairs of 0-terminated wide character string text. These members are only valid for SWItts_cbLogError messages.
defaultMessage	A pre-formatted 0-terminated wide character message. This member is only valid for SWItts_cbDiagnostic messages.

The **bookmark data structure** is described here:

```
typedef struct {
const SFT_TCHAR *        ID;
SSFT_U32                 sampleNumber;
} SWIttsBookMark;
```

Structure members:

ID	A pointer to the bookmark 0-terminated wide character string. It corresponds to the user-defined integer specified in the bookmark tag.
sampleNumber	The bookmark location, specified by an accumulated number of samples for the current SWIttsSpeak( ) call. A bookmark placed at the beginning of a string has a timestamp of 0. The sampleNumber always refers to a sample number in the future (that has not yet been received).

The **wordmark data structure** is described here:

```
typedef struct {
SSFT_U32                 sampleNumber;
SSFT_U32                 offset;
SSFT_U32                 length;
}SWIttsWordMark;
```

Structure members:

sampleNumber	The sample number correlating to the beginning of this word. The sampleNumber always refers to a sample number in the future (that has not yet been received).
offset	The index into the input text of the first character where this word begins. Starts at zero.
length	The length of the word in characters not bytes.

The **phoneme-mark data structure** is described here:

```
typedef struct {
SSFT_U32                sampleNumber;
const char*             name;
SSFT_U32                duration;
SSFT_U32                stress;
}SWIttsPhonemeMark;
```

Structure members:

sampleNumber	The sample number correlating to the beginning of this phoneme. The sampleNumber always refers to a sample number in the future (that has not yet been received).
name	The name of the phoneme as a NULL-terminated US-ASCII string. (The phoneme names are described in the RealSpeak supplements for each language.)
duration	The length of the phoneme in samples.
stress	Not currently supported, this is always set to 0.

Notes

The callback function is user-defined but is called by the SWItts library, i.e., the user writes the code for the callback function, and a pointer to it is passed into the SWIttsOpenPortEx( ) function. The SWItts API library calls this function as needed when data arrives from the RealSpeak engine. It is called from a thread created for the port during the SWIttsOpenPortEx( ) function.
The SWItts_cbStatus variable indicates the reason for invoking the callback and also what, if any, type of data is being passed. The SWIttsResult code returned by the callback is not currently interpreted by RealSpeak, but may be in the future, thus the callback function should always return SWItts_SUCCESS.
Because the callback function is user-defined, the efficiency of its code has a direct impact on system performance - if it is inefficient, it may hinder the SWItts API library's ability to service the engine or TTS server's network connection and data may be lost.
The RealSpeak engine usually delivers audio data to the application faster then real-time. This means that sending a large amount of text to the SWIttsSpeak( ) function may cause the engine to send back a large amount of audio before the application needs to send it to an audio device or telephony card.
On average for Western languages, expect about one second of audio for every ten characters of text input. For example, if you pass 10 KB of text to the SWIttsSpeak( ) function, your callback may receive about 1000 seconds of audio samples. That is 8 MB of data if you chose to receive 8-bit -law samples and 16 MB of data if you chose to receive 16-bit linear samples. This amount of text may require more buffering than you want to allow for, especially in a scenario with multiple TTS ports.

A common technique to avoid a buffering load is to call SWIttsPause( ) when the audio buffer exceeds some application defined buffer size limit (commonly referred to as a "high watermark"), and then call SWIttsResume( ) when the audio buffer falls below some application defined buffer size limit (commonly referred to as a "low watermark").

SWIttsClosePort( )

Mode

Asynchronous

Purpose

Closes a TTS port, freeing all resources and closing all communication with the TTS engine instance.

```
SWIttsResult SWIAPI SWIttsClosePort (
SWIttsPort                    ttsPort
);
```

Parameters

ttsPort	Port handle returned by SWIttsOpenPortEx( ).

After closing, SWIttsClosePort( ) sends a SWItts_cbPortClosed message to the callback upon successful closing of the port. Once a port is closed, you cannot pass that port handle to any SWItts function.

See also

SWIttsOpenPortEx( )

SWIttsDeleteDictionaryEntry( )

Mode

Synchronous

Purpose

Deletes a list of dictionary entries from a dictionary.

```
SWIttsResult SWIAPI SWIttsDeleteDictionaryEntry (
SWIttsPort                ttsPort,
const char*               dictionaryType,
const char*               charset,
SSFT_U32                  numEntries,
SWIttsDictionaryEntry*    entries
);
```

Notes

Not currently supported, always returns SWItts_UNSUPPORTED.
Use SWIttsDictionaryLoad( ) and SWIttsDictionaryFree( ) instead.

SWIttsDictionaryActivate( )

Mode

Synchronous

Purpose

Activate a dictionary for subsequent SWIttsSpeak( ) requests.

```
SWIttsResult SWIAPI SWIttsDictionaryActivate (
SWIttsPort                         ttsPort,
const SWIttsDictionaryData*        dictionary,
SSFT_U32                           priority
);
```

Parameters

ttsPort	Porthandle returned by SWIttsOpenPortEx( )
dictionary	Object containing the URI and fetch parameters, or a string
priority	Priority to assign to this dictionary compared to other active dictionaries. Values: Integers 1–2^31. Lowest priority: 1.

See SWIttsDictionaryLoad( ) for the specification of the SWIttsDictionaryData data structure.

Applications must use SWIttsDictionaryLoad( ) to load a dictionary before activating it.

Activating the dictionary never triggers a reload of the dictionary. To refresh a loaded dictionary that may be changed, call SWIttsDictionaryFree( ) followed by SWIttsDictionaryLoad( ), and then activate the dictionary. See “SWIttsDictionaryLoad( )” for more information.

If you want dictionaries to be active for a speak request, load and activate them before calling SWIttsSpeak( ). (SWIttsSpeak( ) does not require any activated dictionaries.) Once activated, dictionaries are active until they are explicitly deactivated. More than one dictionary can be activated at any given time. When you are finished using all dictionaries, call SWIttsDictionariesDeactivate( ) and then SWIttsDictionaryFree( ) to clean up the resources.

The dictionary priority is a unique integer ranking the priority of this dictionary compared to all other activated dictionaries of the same language and type. During speak requests, the engine performs a lookup in the dictionary of the appropriate type with the highest priority. If the lookup fails, it tries the dictionary of the appropriate type with the next highest priority, until there are no more dictionaries of that type to try.

SWIttsDictionaryActivate( ) may return the following error codes:

SWItts_DICTIONARY_ACTIVE	The dictionary is already activated
SWItts_DICTIONARY_NOT_LOADED	The dictionary is not loaded
SWItts_DICTIONARY_PRIORITY_ALREADY_EXISTS	No duplicate priorities are allowed in dictionaries of the same type and language
SWItts_INVALID_PARAMETER	The ttsPort or dictionary parameter is NULL or invalid
SWItts_MUST_BE_IDLE	A speak operation is active on this port

See also

“SWIttsDictionariesDeactivate( )”
“SWIttsDictionaryFree( )”
“SWIttsDictionaryLoad( )”

SWIttsDictionariesDeactivate( )

Mode

Synchronous

Purpose

Deactivates all activated dictionaries for subsequent speak requests.

```
SWIttsResult SWIAPI SWIttsDictionariesDeactivate (

SWIttsPort                    ttsPort
);
```

Parameters

ttsPort	Port handle returned by SWIttsOpenPortEx( ).

When you are finished using a dictionary, call SWIttsDictionaryFree( ) to clean up the resources used by the dictionary data.

SWIttsDictionariesDeactivate( ) does not deactivate the default dictionaries that are configured for the Realspeak engine.

Active dictionaries remain active until they are explicitly deactivated by SWIttsDictionariesDeactivate( ). They are not automatically deactivated after each speak request. SWIttsDictionariesDeactivate( ) deactivates all active dictionaries. There is no way to deactivate individual dictionaries. To deactivate only some of the currently active dictionaries, use this function to deactivate all dictionaries, then re-activate the desired dictionaries with SWIttsDictionaryActivate( ).

SWIttsDictionariesDeActivate( ) may return the following error codes:

SWItts_INVALID_PARAMETER	The ttsPort parameter is NULL or invalid
SWItts_MUST_BE_IDLE	A speak operation is active on this port

See also

“SWIttsDictionaryActivate( )”
“SWIttsDictionaryFree( )”
“SWIttsDictionaryLoad( )”

SWIttsDictionaryFree( )

Mode

Synchronous

Purpose

Signals the engine that the dictionary is no longer needed.

```
SWIttsResult SWIAPI SWIttsDictionaryFree (
SWIttsPort                       ttsPort,
const SWIttsDictionaryData*      dictionary
);
```

Parameters

ttsPort	Porthandlereturned by SWIttsOpenPortEx( ).
dictionary	Object containing the URI and fetch parameters, or a string.

When you are finished using a dictionary, call SWIttsDictionaryFree( ) to clean up the resources used by the dictionary data.

SWIttsDictionaryFree( ) *cannot* be used to free the default dictionaries that are configured for the RealSpeak engine.

SWIttsDictionaryFree( ) may return the following error codes:

SWItts_DICTIONARY_ACTIVE	Dictionary cannot be freed until deactivated.
SWItts_DICTIONARY_NOT_LOADED	Dictionary is not loaded
SWItts_INVALID_PARAMETER	The ttsPort or dictionary parameter is NULL or invalid.
SWItts_MUST_BE_IDLE	A speak operation is active on this port.

See also

“SWIttsDictionaryActivate( )”
“SWIttsDictionariesDeactivate( )”
“SWIttsDictionaryLoad( )”

SWIttsDictionaryLoad( )

Mode

Synchronous

Purpose

Load a complete dictionary from a URI or string to prepare it for future activation.

```
SWIttsResult SWIAPI SWIttsDictionaryLoad (
SWIttsPort                        ttsPort,
constSWIttsDictionaryData*        dictionary
);
```

Parameters

ttsPort	Port handle returned by SWIttsOpenPortEx( )
dictionary	Object containing the URI and fetch parameters, or a string

A dictionary must be loaded before it can be activated. If you want the engine to apply dictionaries to text passed in the SWIttsSpeak( ) and SWIttsSpeakEx( ) functions, you must load and activate them before calling SWIttsSpeak( ) or SWIttsSpeakEx( ).

SWIttsDictionaryLoad( ) blocks until dictionary loading and parsing is complete.

Structures

The **SWIttsDictionaryData structure** is defined as follows:

```
typedef struct SWIttsDictionaryData {
SSFT_U32                    version;
const char*                 uri;
const unsignedchar*         data;
SSFT_U32                    lengthBytes;
const char*                 contentType;
const VXIMap*               fetchProperties
VXIMap*                     fetchCookieJar;
}SWIttsDictionaryData;
```

Structure members:

version	Use the constant SWItts_CURRENT_VERSION, defined in SWItts.h.
uri	URI to the dictionary content; *contentType* may be NULL. Pass NULL when *data* is non-NULL. The URI may be one of the following: * (RealSpeak Telecom only) HTTP/1.1 web server access, where the URL is fetched by the Realspeak server: http: //myserver/mydict. xml * Simple file access. For RealSpeak Telecom, the file is resolved on the Realspeak server. for example: file:/users/mydict.xml /users/mydict.xml
data	In-memory dictionary content; *contentType* must be non-NULL. Pass NULL when *uri* is non-NULL
length Bytes	Length of the in-memory dictionary content in bytes. Pass 0 when *uri* is non-NULL
content Type	MIME content type to identify the dictionary format. One of the following: * NULL: only valid when type is "uri". Automatically determines the content type from the URI. For http: URIs, the MIME content type returned by the web server is processed using the rules that follow. For file: URIs, files with a .xml extension are treated as Scansoft dictionaries, otherwise an error results * application/octet-stream: assume a RealSpeak dictionary (this is the default MIME content type returned by web servers for unknown data types) * application/edct-bin-dictionary: RealSpeak binary format dictionary * application/edct-text-dictionary: RealSpeak text format dictionary * text/xml: assume a RealSpeak text format dictionary for backward compatibility with Speechify (this permits migrating the dictionary in-place without changing C code) application/x-swi-dictionary: assume a RealSpeak text format dictionary for backward compatibility with Speechify (this permits migrating the dictionary in-place without changing C code) Other: a SWItts_INVALID_MEDIATYPE error is returned
fetchProperties	(RealSpeak Telecom only) Optional VXIMap to control Internet fetch operations (particularly the base URI and fetch timeouts). May be NULL to use defaults. These settings apply to the fetch of the dictionary when uri is

	non-NULL.
fetchCookieJar	(RealSpeak Telecom only) Optional VXIMap to provide session or end-user-specific cookies for Internet fetch operations, modified to return the updated cookie state on success. May be NULL to disable cookies (web server cookies are refused).

If an application asks SWIttsDictionaryLoad( ) to load a dictionary that is already loaded, Realspeak returns the non-fatal error code SWItts_DICTIONARY_ LOADED. To refresh Realspeak's copy of a dictionary that has been updated or changed elsewhere, call SWIttsDictionaryFree( ) then SWIttsDictionaryLoad( ) to force Realspeak to reload the dictionary.

SWIttsDictionaryLoad( ) may return the following error codes:

SWItts_DICTIONARY_LOADED	Dictionary is already loaded.
SWItts_DICTIONARY_PARSE_ ERROR	Any error during dictionary parsing.
SWItts_INVALID_MEDIATYPE	Unsupported MIME content type for the dictionary format.
SWItts_INVALID_PARAMETER	The ttsPort or dictionary parameter is NULL or invalid.
SWItts_MUST_BE_IDLE	A speak operation is active on this port
SWItts_UNKNOWN_CHARSET	Character set for the dictionary is invalid or unsupported
SWItts_URI_FETCH_ERROR	Any error during URI access other than SWItts_ URI_NOT_FOUND or SWItts_URI_ TIMEOUT
SWItts_URI_NOT_FOUND	URI was not found (file does not exist or the web server does not have a matching URI).
SWItts_URI_TIMEOUT	Timeout during web server URI access

See also

“SWIttsDictionaryActivate( )”
“SWIttsDictionariesDeactivate( )”
“SWIttsDictionaryFree( )”

SWIttsGetDictionaryKeys( )

Mode

Synchronous

Purpose

Enumerates dictionary keys from the specified dictionary.

```
SWIttsResult SWIAPI SWIttsGetDictionaryKeys(
SWIttsPort                      ttsPort,
const char*                     dictionaryType,
SWIttsDictionaryPosition*       startingPosition,
SSFT_U32*                       numkeys,
SWIttsDictionaryEntry**         keys,
const char*                     reserved
);
```

Notes

Not currently supported, always returns SWItts_UNSUPPORTED.

SWIttsGetParameter( )

Mode

Synchronous

Purpose

Retrieves the value of a parameter from the server.

```
SWIttsResult SWIAPI SWIttsGetParameter (
SWIttsPort                    ttsPort,
const char*                   name,
char*                         value
);
```

Parameters

ttsPort	Theporthandle returned by SWIttsOpenPortEx().
name	The name of the parameter to retrieve.
value	Takes a preallocated buffer of size SWITTS_MAXVAL_SIZE.

The following table describes the parameters that can be retrieved. Certain parameters are read-only.

Name	Possible values	Read -only	Description
tts.audio.packetsize	even number 64-102400	no	Maximum size of the audio packets, in bytes, that the SWItts API sends to the user supplied callback function. (All packets will be this size except the last packet, which may be smaller.)
tts.audio.rate	33-300	no	Port-specific speaking rate of the synthesized text as a percentage of the default rate.
tts.audio.volume	0-100 (see description for caveat.)	no	Port-specific volume of synthesized speech as a percentage of the default volume: 100 means maximum possible without distortion and 0 means silence. Values greater than 100 are permitted, but output might have distortion.
tts.audioformat.encoding	ulaw, alaw, linear	yes	Encoding method for audio generated during synthesis. This value can be set via the mimetype.
tts.audioformat.mimetype	audio/basic audio/x-alaw-basic audio/L16;rate=8000	No	The audio format:

	audio/L16;rate=11000 audio/L16;rate=16000 audio/L16;rate=22000		* audio/basic corresponds to 8 kHz, 8-bit -law; * audio/x-alaw-basic corresponds to 8 kHz, 8-bit A-law; * audio/L16;rate=8000 corresponds to 8 kHz, 16-bit linear; * audio/L16;rate=11000 corresponds to 11 kHz, 16-bit linear; * audio/L16;rate=16000 corresponds to 16 kHz, 16-bit linear. * audio/L16;rate=22000 corresponds to 22 kHz, 16-bit linear; All other values generate a SWItts_INVALID_PARAMETER return code. In all cases, audio data is returned in network byte order.
tts.audioformat.samplerate	8000, 11000, 16000, 22000	yes	Audio sampling rate in Hz. This value can be set via the mimetype
tts.audioformat..width	8, 16	yes	Size of individual audio sample in bits This value can be set via the mimetype
tts.client.version	Current RealSpeak SWItts API library version number	yes	The returned value is a string of the form major.minor.maintenance. For example, 2.0.0 or 2.0.1. This parameter reflects the SWItts API library version, and can be retrieved after SWIttsInit( ) is called but before SWIttsOpenPortEx( ) is called. Use SWITTS_INVALID_PORT for the first argument to SWIttsGetParameter( ).
tts.engine.id	positive integer	yes	RealSpeak engine logical channel ID that is handling speak requests for this SWItts API library port. The RealSpeak engine logs diagnostics, errors, and events to its diagnostic and error log using this logical channel ID, so including this logical channel ID in application logs can help in cross-referencing application and RealSpeak engine logs.

tts.engine.version	Current Realspeak engine version number	yes	The returned value is a string of the form major.minor.maintenance. For example, 2.0.0 or 2.0.1.
tts.marks.phoneme	true, false	no	controls whether phoneme marks are reported to the client
tts.mark.word	true, false	no	controls whether wordmarks are reported to the client
tts.network.timeout	positive integer	no	(RealSpeak Telecom client/server mode only) Timeout, in seconds, for the connection to the RealSpeak server. If a send operation to the server fails to complete within this duration, or if a heartbeat is not received from a server in this duration, the server connection is presumed to be dead and the connection is dropped.
tts.product.name	"RealSpeak Host", "RealSpeak Solo"	yes	"RealSpeak Host" for the main RealSpeak Host product; "RealSpeak Solo" for the RealSpeak Solo product.

tts.server.licensingMode	default, explicit	yes	Modes for controlling license allocation to a Realspeak port object * default: automatically when SWIttsOpenPortEx() is called * explicit: as decided by the platform developer. Use SWIttsResourceAllocate() and SWIttsResourceFree() to control allocation and de-allocation of licenses
tts.voice.gender	male, female	yes	synthesis voice gender
tts.voice.language	server	yes	synthesis language
tts.voice.name	server	yes	unique name identifying the voice
tts.voice.version	current Realspeak voice version number	yes	The returned value is a string of the form major.minor.maintenance. For example, 2.0.0.

See also

“SWIttsSetParameter( )”

SWIttsInit( )

Mode

Synchronous

Purpose

Initializes the SWItts API library so that it is ready to open ports.

```
SWIttsResult SWIAPI SWIttsInit (
SWIttsCallback*          callback,
SWIttsCallback*          userData
);
```

Parameters

callback	A pointer to a callback function that may receive SWItts_cbLogError and/or SWItts_cbDiagnostic messages during the SWIttsInit( ) call. If this callback is called, the ttsPort parameter is –1. This may be the same callback that is passed to SWIttsOpenPortEx( ) or SWIttsTerm( ).
userData	User information passed back to callback. It is not interpreted or modified in any way by the SWItts API library

Notes

This must be the first API function called, and it should only be called once per process, not once per call to SWIttsOpenPortEx( ).

SWIttsInit( ) may return the following error codes:

SWItts_ALREADY_INITIALIZED	The SWItts API library is already initialized (from a prior SWIttsInit( ) call).

See also

“SWIttsOpenPortEx( )”
“SWIttsTerm( )”

SWIttsLookupDictionaryEntry( )

Mode

Synchronous

Purpose

Retrieves the translation for the given key from the specified dictionary.

```
SWIttsResult SWIAPI SWIttsLookupDictionaryEntry(
SWIttsPort                  ttsPort,
const char*                 dictionaryType,
const SSFT_U8*              key,
const char*                 charset,
SSFT_U32                    keyLengthBytes,
SSFT_U32*                   numEntries,
SWIttsDictionaryEntry**     entries
);
```

Notes

Not currently supported, always returns SWItts_UNSUPPORTED.

SWIttsOpenPort()

Mode

Synchronous

Purpose

This function opens and connects to a RealSpeak engine port. Call this function after SWIttsInit( ).

RealSpeak Solo:

```
SWIttsOpenPort(
SWIttsPort*                    ttsPort,
const char*                    parameters,
SWIttsCallback*                callback,
void*                          userData
);
```

RealSpeak Telecom:

```
SWIttsResult SWIAPI SWIttsOpenPort(SWIttsPort *ttsPort,
const char*                    hostAddr,
SSFT_U16                       connectionPort,
SWIttsCallback*                callback,
void*                          userData
);
```

Notes

SWIttsOpenPort() is merely a Speechify 2.1 and Speechify Solo 1.0 compatibility layer on top of SWIttsOpenPortEx( ). See SWIttsOpenPortEx( ) for parameter and return code details.

For **RealSpeak Solo**, SWIttsOpenPort( ) is equivalent to calling:

```
SWIttsOpenPortEx(ttsPort, parameters, NULL, callback, userData);
```

For **RealSpeak Telecom**, SWIttsOpenPort( ) is equivalent to:

```
char parameters[1024];
sprintf(parameters, “hostname=%s;hostport=%u”, hostAddr,
connectionPort);
SWIttsOpenPortEx(ttsPort, parameters, NULL, callback, userdata);
```

See also

SWIttsOpenPortEx( )
SWIttsClosePort( )
SWIttsInit( )

SWIttsOpenPortEx( )

Mode

Synchronous

Purpose

This function opens and connects to a Realspeak engine port. Call this function after SWIttsInit( ).

```
SWIttsOpenPortEx(
SWIttsPort*              ttsPort,
const char*              parameters,
SWIttsResources*         resources,
SWIttsCallback*          callback,
void*                    userdata
);
```

Parameters

ttsPort	Addressofa location to place the new port's handle
parameters	Key/value parameter list in <key1>=<value1>;<key2>=<value2>[...] form. If there is no engine matching the specified parameters, or if the set of parameters does not uniquely identify an engine, the call returns SWItts_INVALID_PARAMETER. Use these keys and values in the parameters field to specify a voice: • language, such as "American English" • name, such as "Jennifer" • sample_rate, such as "8000" • (RealSpeak Solo only) quality, currently ignored • (RealSpeak Solo only) reduction_type, a voice reduction type as listed in the RealSpeak language documentation • (RealSpeak Telecom only) hostname, a RealSpeak server host name such as "localhost" • (RealSpeak Telecom only) hostport, a RealSpeak server host TCP/IP port number such as "6666"
resources	Reserved for future use, pass NULL
callback	A pointer to a callback function that receives audio buffers and other notifications when the server sends data to the client. If an error occurs during the call to SWIttsOpenPortEx( ), the callback is called with a SWItts_cbLogError message and a ttsPort of –1.
userData	User information passed back to callback

Notes

For compatibility with Speechify, RealSpeak Telecom applies special mapping rules if hostname and hostport are specified without specifying a language or name. These mapping rules are loaded from config/swittsclient.cfg when SWIttsInit( ) is called, and is used to translate a Speechify port number to a RealSpeak port number, language name, voice name, and sample rate. This is required because a Speechify server instance (hostname and hostport pair) could only support one voice and sample rate, so that information was sufficient to also identify the desired language, voice, and sample rate. However, a RealSpeak server instance can support any number of languages, voices, and sample rates simultaneously. If you are using the default Speechify and RealSpeak server port numbers, then the default mapping rules should be sufficient. If you use custom server port numbers, however, then you should customize swittsclient.cfg to define rules for mapping those port numbers. See swittsclient.cfg for details.

SWIttsOpenPortEx( ) may return the following error codes:

SWItts_INVALID_PARAMETER	One of the parameters to the function was invalid.
SWItts-NO_LICENSE	There are no purchased licenses available

Example

Below is an example of how you would use this API function for **RealSpeak Solo:**

```
SWIttsPort port;
SWIttsOpenPortEx(&port,
"language=American English;name=Jennifer;sample_rate=8000",
callback, NULL);
```

Below is an example of how you would use this API function for **RealSpeak Telecom:**

```
SWIttsPort port;
SWIttsOpenPortEx(&port,
"hostname=localhost;hostport=6666;language=American
English;name=Jennifer;sample_rate=8000",
callback, NULL);
```

See also

“SWIttsClosePort( )”
“SWIttsInit( )

SWIttsPause( )

Mode

Asynchronous

Purpose

(RealSpeak Solo only) Pauses the current active speak request.

```
SWIttsResult SWIAPI SWIttsPause (
SWIttsPort                    ttsPort
);
```

Parameters

ttsPort	The port handle returned by SWIttsOpenPortEx( ).

b)

Notes

This pauses the delivery of audio, marks, and other events for the current active speak request. To resume the request, call SWIttsResume( ). Note that since RealSpeak usually delivers audio faster then real-time, this call is not sufficient to fully implement an application level pause operation: to fully implement a pause for the end user, the application needs to pause the audio playback device, then call this API function to prevent overflowing the application level audio buffer.
If there is no SWIttsSpeak( ) function in progress, or if a currently active speak request is already paused due to a previous call to SWIttsPause( ), this function returns an error.

See also

SWIttsSpeak( )
SWIttsSpeakEx( )
SWIttsResume( )

SWIttsPing( )

Mode

Asynchronous

Purpose

Performs a basic test of the TTS engine instance responsiveness.

```
SWIttsResult SWIAPI SWIttsPing (
SWIttsPort                    ttsPort
);
```

Parameters

ttsPort	The port handle returned by SWIttsOpenPort( )

This verifies that the instance of the TTS engine instance for this port is alive and accepting requests.
A return code of SWItts_SUCCESS means that the ping has been successfully sent to the TTS port. When the TTS engine instance replies, the SWItts API library calls the callback for this port with a status of SWItts_cbPing. If this function returns an error code, shut down the port with the SWIttsClosePort( ) call. The amount of time you should wait for the SWItts_cbPing message in your callback varies depending on the load on your system; a good rule of thumb is to wait about five seconds for a ping reply before assuming the port is dead.

See also

SWIttsClosePort( )
SWIttsOpenPort( )

SWIttsResetDictionary( )

Mode

Synchronous

Purpose

Removes all entries from the specified dictionary.

```
SWIttsResult SWIAPI SWIttsResetUserDictionary(
SWIttsPort                ttsPort,
const char*               dictionaryType
);
```

Notes

Not currently supported, always returns SWItts_UNSUPPORTED.

SWIttsResourceAllocate( )

Purpose

(RealSpeak Telecom only) Explicitly assign a license for a specified Realspeak port.

```
SWIttsResult SWIttsResourceAllocate(
SWIttsPort              ttsPort,
const SSFT_TCHAR*       feature,
void*                   reserved
);
```

Parameters

ttsPort	Theporthandlereturned by SWIttsOpenPortEx( ).
feature	Use the constant SWItts_LICENSE_SPEAK defined in SWItts.h for licensing functionality.
reserved	This parameter is reserved for future use. Pass in NULL

Notes

The tts.server.licensingMode configuration parameter must be set to "explicit" for SWIttsResourceAllocate( ) to work. You can use SWIttsGetParameter( ) to retrieve the value of tts.server.licensingMode and find out whether you need to call this function (and explicitly allocate and free licenses) or not. If the licensing mode is set to "default," the SWIttsOpenPortEx( ) function implicitly allocates a license for the application.

SWIttsResourceAllocate( ) may return the following error codes:

SWItts_INVALID PARAMETER	An invalid feature parameter was specified
SWItts_LICENSE_ALLOCATED	A license has already been allocated for this port..
SWItts_MUST_BE_IDLE	A speak operation is active
SWItts_NO_LICENSE	There are no purchased licenses available
SWItts_UNSUPPORTED	The tts.server.licensingMode parameter is not set to explicit.

See also

"SWIttsResourceFree( )"

SWIttsResourceFree( )

Purpose

(RealSpeak Telecom only) Explicitly free the user license for the specified Realspeak port.

```
SWIttsResult SWIttsResourceFree(
SWIttsPort                    ttsPort,
const SSFT_TCHAR*             feature,
void*                         reserved
);
```

Parameters

ttsPort	Theporthandlereturnedby SWIttsOpenPortEx( ).
feature	Use SWItts_LICENSE_SPEAK to free a license
reserved	This parameter is reserved for future use. Pass in NULL.

Notes

The tts.server.licensingMode configuration parameter must be set to explicit for SWIttsResourceFree( ) to work.

Note that SWIttsClosePort( ) also frees the license for a port while freeing all other resources for that port.

SWIttsResourceFree( ) may return the following error codes:

SWItts_INVALID_PARAMETER	An invalid feature parameter was specified
SWItts_LICENSE_FREED	A license has already been freed for this port.
SWItts_MUST_BE_IDLE	A speak operation is active.
SWItts_NO_LICENSE	There are no purchased licenses available
SWItts_UNSUPPORTED	The tts.server.licensingMode parameter is not set to explicit

See also

“SWIttsClosePort( )
“SWIttsResourceAllocate( )”

SWIttsResume( )

Mode

Asynchronous

Purpose

(RealSpeak Solo only) Resumes a paused active speak request.

```
SWIttsResult SWIAPI SWIttsResume (
SWIttsPort                    ttsPort
);
```

Parameters

ttsPort	The port handle returned by SWIttsOpenPortEx( ).

Notes

This resumes the delivery of audio, marks, and other events for a paused speak request.
If there is no SWIttsSpeak( ) function in progress, or if a currently active speak request is not paused, this function returns an error.

See also

SWIttsSpeak( )
SWIttsSpeakEx( )
SWIttsPause( )

SWIttsSetParameter( )

Mode

Synchronous

Purpose

Sends a parameter to the TTS engine instance.

```
SWIttsResult SWIAPI SWIttsSetParameter(
SWIttsPort                    ttsPort,
const char*                   name,
const char*                   value
);
```

ttsPort	The port handle returned by SWIttsOpenPortEx( ).
name	A parameter name represented as a NULL-terminated US-ASCII string
value	A parameter value represented as a NULL-terminated US-ASCII string

Notes

If SWIttsSetParameter( ) returns an error, the parameter is not changed. Setting a parameter is not a global operation, it only affects the TTS port passed to the call.

The following table describes the parameters that can be set. All parameters have a default value from the server XML configuration file. SWIttsGetParameter( ) lists the read-only parameters. If you try to set a read-only parameter, SWIttsSetParameter( ) returns SWItts_READ_ONLY.

Name	Possible values	Description
tts.audio.packetsize	Even number 64-102400 Recommended values: 1024,2048,or 4096	Maximum size of the audio packets, in bytes, that the SWItts API sends to the user supplied callback function. (All packets are this size except the last packet, which may be smaller.)
tts.audio.rate	33-300	Port-specific speaking rate of the synthesized text as a percentage of the default rate.
tts.audio.volume	0-100 (See description for caveat.)	Port-specific volume of synthesized speech as a percentage of the default volume: 100 means maximum possible without distortion and 0 means silence. Values greater than 100 are permitted, but output might have

		distortion.
tts.audioformat.mimetype	audio/basic audio/x-alaw-basic audio/L16;rate=8000 audio/L16;rate=11000 audio/L16;rate=16000 audio/L16;rate=22000	The audio format of the server - audio/basic corresponds to 8 kHz, 8-bit μ-law - audio/x-alaw-basic corresponds to 8 kHz, 8-bit A-law - audio/L16;rate=8000 corresponds to 8 kHz, 16-bit linear - audio/L16;rate=11000 corresponds to 11 kHz, 16-bit linear - audio/L16;rate=16000 corresponds to 16 kHz, 16-bit linear - audio/L16;rate=22000 corresponds to 22 kHz, 16-bit linear
		All other values generate a SWItts_INVALID_PARAM return code.
		In all cases, audio data is returned in network byte order.
tts.marks.phoneme	true, false	Controls whether phoneme marks are reported to the application.
tts.marks.word	true, false	Controls whether wordmarks are reported to the application.
tts.network.timeout	positive integer	(RealSpeak Telecom client/server mode only) Timeout, in seconds, for the connection to the RealSpeak server. If a send operation to the server fails to complete within this duration, or if a heartbeat is not received from a server in this duration, the server connection is presumed to be dead and the connection is dropped.
tts.reset	none	Command which causes all parameters controllable via SWIttsSetParameter( ) to revert to their default values; the value is ignored.

tts.audioformat.mimetype values may be switched between audio/basic, audio/x-alaw-basic, and audio/L16;rate=8000 if the server has been instantiated with the 8 kHz voice database. If the server is instantiated with the 1 6 kHz voice database, this parameter has the read-only value of audio/L16:rate=16000

In a client/server environment, the default rate and volume is set in RealSpeak Server configuration file (ttsserver.xml, see "Configuration Files" section in the "User Configuration" chapter). If the rate or

volume is set through this API call, the new value overrides those defaults. Similarly, if the rate or volume is set through markup in the input text, those values override both the RealSpeak Server default and the value set via the API for that (and only that) speak request.

See also

"SWIttsGetParameter( )"

SWIttsSpeak( )

Mode

Asynchronous

Purpose

Sends a text string to be synthesized. Call this function for every text string to synthesize.

```
SWIttsResultSWIAPI        SWIttsSpeak(
SWIttsPort                ttsPort,
const SSFT_U8*            text,
SSFT_U32                  lengthBytes,
const char*               content_type
);
```

Parameters

ttsPort	The port handle returned by SWIttsOpenPortEx( )
text	The text to be synthesized: an array of bytes representing a string in a given character set.
lengthBytes	The length of the text array in bytes; note that this means any NULL in the text is treated as just another character.
content_type	Description of the input text according to the MIME standard (per RFC-2045 Sec. 5.1 and RFC 2046). Default (if set to NULL): text/plain;charset=iso-8859-1.

Notes

See SWIttsSpeakEx( ) to do either of these tasks:

- Have Realspeak fetch the document to speak from a web server (instead of the text being in memory)
- (RealSpeak Telecom only) Specify internet fetch controls for <audio> elements within W3C SSML documents

The content types that are supported are text/* and application/ssml+xml (or application/synthesis+ssml).

Any subtype may be used with "text". However, only the subtype "xml" is treated specially: the text is assumed to be in W3C SSML and if it is not, an error is returned. All other subtypes are treated as "plain".
(RealSpeak Telecom only) The "application/ssml+xml" content type is used to indicate W3C SSML content, which is parsed accordingly. If W3C SSML input is not signaled via the content type parameter, it is pronounced as plain text.
The only meaningful content_type parameter is "charset," which is case-insensitive. (See *www.iana.org/assignments/character-sets* for more

details.) All other parameters are ignored. If "charset" is not specified, it is assumed to be ISO-8859-1. An example of a valid content type:

- text/plain;charset=iso-8859-1

The supported character sets vary by language:

Character set	Languages	Notes
UTF-8	All languages	
UTF-16	All languages	If the byte order mark is missing, big-endian is assumed
wchar_t	All languages	"wchar_t" is not a MIME standard. It indicates that the input is in the form of the operating system's native wide character array (i.e., wchar_t *). Note that input length must still be specified in bytes (i.e., the number of wide characters in the input times the number of bytes per wide character).
ISO-8859-1	Western languages	
US-ASCII (synonym: ASCII)	Western languages	
windows-1252	Western languages	
EUC-jp (synonym: EUC)	Japanese	
Shift-JIS	Japanese	

SWIttsSpeak( ) may return the following error codes:

SWItts_NO_LICENSE There are no purchased licenses available

See also

"SWIttsSpeakEx( )"
"SWIttsStop( )"

SWIttsSpeakEx( )

Mode

Asynchronous

Purpose

Sends a text URI or string to be synthesized. Call this function for every text URI or string to synthesize.

```
SWIttsResult SWIAPI SWIttsSpeakEx(
SWIttsPort                    ttsPort,
const SWIttsSpeakData*        speakData
);
```

Parameters

ttsPort	The port handle returned by SWIttsOpenPortEx( ).
speakData	Object containing the URI and fetch parameters, or a string.

Structures

The **SWIttsSpeakData** structure is defined as follows:

```
Typedef struct SWIttsSpeakData{
SSFT_U32                      version;
const char*                   uri;
const SSFT_U8*                data;
SSFT_U32                      lengthBytes;
const char*                   contentType;
const VXIMap*                 fetchProperties;
VXIMap*                       fetchCookieJar;
}SWIttsSpeakData;
```

Structure members:

version	Use the constant SWItts_CURRENT_VERSION defined in SWItts.h.
uri	URI to the text; *contentType* may be NULL. Pass NULL when *data* is non-NULL. The URI may be one of the following: * HTTP/1.1 web server access, where the URL is fetched by the Realspeak server: http://myserver/mytext.txt * Simple file access. For RealSpeak Telecom, the file is resolved on the Realspeak server. For example: file: /users/mytext. txt /users/mytext.txt
data	In-memory text; *contentType* must be non-NULL. Pass NULL when *uri* is non-NULL.
lengthBytes	Length of the in-memory text in bytes. Pass 0 when *uri* is

	non-NULL
contentType	MIME content type to identify the text format. One of the following: * NULL: only valid when type is "uri". Automatically determines the content type from the URI. For http:// URIs, the MIME content type returned by the web server is processed using the rules that follow. For file: URIs, files with a .dct extension are treated as W3C SSML documents, and files with a .txt extension are treated as ISO-8859-1 text documents, otherwise an error results. * text/*: Subtype xml text is assumed to be in W3C SSML. If it is not, an error is returned. Other subtypes are treated as "plain." * (RealSpeak Telecom only) application/ssml+xml: indicates W3C SSML content. If W3C SSML input is not indicated via contentType, it is pronounced as plain * Other: a SWItts_INVALID_MEDIATYPE error is returned
fetchProperties	(RealSpeak Telecom only) Optional VXIMap to control Internet fetch operations (particularly the base URI and fetch timeouts). May be NULL to use defaults. These settings apply to the fetch of the top-level document when *uri* is non-NULL, and also to any fetches for <audio> elements within W3C SSML documents (whether the W3C SSML document was fetched by URI or provided in-memory).
fetchCookieJar	(RealSpeak Telecom only) Optional VXIMap to provide session or end-user-specific cookies for Internet fetch operations, modified to return the updated cookie state on success. May be NULL to disable cookies (web server cookies are refused).

The only meaningful contentType parameter is "charset," which is case-insensitive. (See *www.iana.org/assignments/character-sets* for more details.) All other parameters are ignored. If "charset" is not specified, it is assumed to be ISO-8859-1. An example of a valid contentType:

- text/plain;charset=iso-8859-1

The supported character sets vary by language:

Character set	Languages	Notes
UTF-8	All languages	
UTF-16	All languages	If the byte order mark is missing, big-endian is assumed
wchar_t	All languages	"wchar_t" is not a MIME standard. It indicates that the input is in the form of the operating system's native wide character array (i.e., wchar_t *). Note that input length must still be specified in bytes (i.e., the number of wide characters in the input times the number of

		bytes per wide character).
ISO-8859-1	Western languages	
US-ASCII (synonym: ASCII)	Western languages	
windows-1252	Western languages	
EUC-jp (synonym: EUC)	Japanese	
Shift-JIS	Japanese	

SWIttsSpeakEx( ) may return the following error codes:

SWItts_INVALID_MEDIATYPE	Unsupported MIME content type for the speak data.
SWItts_INVALID_PARAMETER	The ttsPort or speakData parameter is NULL or invalid.
SWItts_MUST_BE_IDLE	A speak operation is active.
SWItts_NO_LICENSE	There are no purchased licenses available.
SWItts_UNKNOWN_CHARSET	Character set for speakData is invalid or unsupported.
SWItts_URI_FETCH_ERROR	Any error during URI access other than SWItts_URI_NOT_FOUND and SWItts_URI_TIMEOUT.
SWItts_URI_NOT_FOUND	URI was not found (file does not exist or the web server does not have a matching URI).
SWItts_URI_TIMEOUT	Timeout during web server URI access.

See also

SWIttsStop( )

SWIttsStop( )

Mode

Asynchronous

Purpose

Interrupts a call to SWIttsSpeak( ).

```
SWIttsResult SWIAPI SWIttsStop (
SWIttsPort ttsPort
);
```

Parameters

ttsPort	The port handle returned by SWIttsOpenPortEx( ).

c)

Notes

When the currently active speak request is completely stopped and the port is idle, the SWItts library calls the port's callback with a status of SWItts_cbStopped. The callback is called with SWItts_cbStopped only if the SWIttsStop( ) function returns with a SWItts_SUCCESS result.
If there is no SWIttsSpeak( ) function in progress, or if a currently active speak request is already stopping due to a previous call to SWIttsStop( ), this function returns an error.

See also

SWIttsSpeak( )
SWIttsSpeakEx( )

SWIttsTerm( )

Mode

Synchronous

Purpose

Closes all ports, terminates their respective threads, shuts down the API library, and cleans up memory usage.

```
SWIttsResult SWIAPI SWIttsTerm(
SWIttsCallback*                    callback,
void*                              userData
);
```

Parameters

callback	A pointer to a callback function that may receive SWItts_cbError, SWItts_cbLogError, and/or SWItts_cbDiagnostic messages during the SWIttsTerm( ) call.
userData	User information passed back to callback.

Notes

If SWIttsTerm( ) closes one or more open TTS ports, you receive SWItts_cbPortClosed messages in their respective callbacks.

See also

SWIttsInit( )

RealSpeak Telecom
Software Development Kit

Chapter XI

Speechify Email Pre-Processor

Programmer's Guide

Chapter XI

Speechify Email Pre-Processor

Introduction

Speechify™ is an older Text-To-Speech (TTS) system that was combined with RealSpeak Solo 1.0 and RealSpeak Telecom 3.5 to create the best-of-breed RealSpeak product you are using now. This chapter is written for Speechify 2.1 and 3.0 application developers who previously used the email pre-processor library of those Speechify products, and wish to continue to use this same email preprocessing solution with the converged RealSpeak products.

If you have not previously used the Speechify 2.1 or 3.0 e-mail pre-processor, however, you should **not** use this email pre-processor. Instead, you should use the email pre-processor that is built-in to the RealSpeak product. See the RealSpeak User's Guides for more information.

- Programming environment.

This paragraph gives an overview of features, how to use the e-mail pre-processor API, and information about the e-mail substitution dictionary, used to specify and customize text substitutions. For a detailed explanation of the API functions, see "API Reference".

Features

The Speechify E-mail Pre-processor is provided as a dynamic linked library (.dll) on NT and a static library (.a) on UNIX. It has the following features:

- Support for standard format e-mail.
- A powerful e-mail substitution dictionary.
- Thread-safe API functions.

The Speechify E-mail Pre-processor only processes plain text e-mail messages. It does not process e-mail messages that are in HTML format or are encoded (e.g., base64). The pre-processor does provide the facility to support partially processed email messages, as well as MIME format. This allows the pre-processor to process e-mail messages that have been pre-filtered for HTML tags (or any other text format that the application chooses to handle). A full description of this is provided in chapter "Functionality of the E-mail Pre-processor"

Chapter XI

The pre-processor has a substitution dictionary that allows the user to specify the way strings are spoken by RealSpeak. Entries in the dictionary tell the pre-processor to substitute certain pieces of text with other text. For example, you may want to replace "BTW" with "by the way". Dictionary entries may apply to the whole message or to specific sections of the message, depending on the scope that the user has specified. A full description of the dictionary is provided in chapter "Using the E-mail Substitution Dictionary".

Order of API calls

There are just three API functions in the e-mail pre-processor. You must call them in this order.

- SWIemailInit( ) initializes the e-mail preprocessor library which includes loading the substitution dictionary and loading it into memory
- SWIemailProcess( ) processes a single e-mail message in the form of a null terminated text string.
- SWIemailTerm( ) frees resources needed by the library.

Pseudo-code for an offline e-mail pre-processor may look like this:

```
SWIemailInit()
while (more messages to process)
        SWIemailProcess(message n)
Save(message n)
SWIemailTerm()
```

Pseudo-code for an application that processes and speaks e-mail at runtime may look like this:

```
SWIemailInit()
SWIttsInit()
SWIttsOpenPort()
while (more TTS requests to make)
        SWIemailProcess ()
        SWIttsSpeak()
SWIttsClosePort()
SWIttsTerm()
SWIemailTerm()
```

NOTE

You only need to call SWIemailInit( ) and SWIemailTerm( ) once per process, and you need to call SWIemailProcess( ) once per message.

Chapter XI

Speechify Email Pre-processor

Chapter XI

Functionality of the E-mail Pre-Processor

This paragraph explains the types of input that the Speechify E-mail Pre-processor handles, how it processes the messages, and the modes that the application can take advantage of at run time. There is mention of MIME (Multipurpose Internet Mail Extensions) format messages below. A MIME format message is one that conforms to the Internet standards defined in RFCs 822, 2045 and 2046. (http://www.ietf.org/rfc.html)

In This Paragraph

- Supported message formats
- Default behavior
- Modes

Chapter XI

Supported message formats

The Speechify E-mail Pre-processor deals with messages in full MIME format, or partially processed messages. It only processes plain text e-mail messages. It does not process e-mail messages that are in HTML format or that are encoded (e.g., base64), therefore HTML/XML tags or encoded text should be filtered out of the message or decoded before a call to SWIemailProcess( ). If not, RealSpeak reads the tags or encoded text. If you preprocess a message with another application (e.g., to parse or filter HTML tags), and the output is not in MIME format, then the e-mail preprocessor can still deal with the message. The only assumption is that there is an empty line separating the header from the body of the message. Below is an example of a partially processed e-mail.

```
From: "Dave Burns" <david.burns@scansoft.com>
To: "Daniel Faulkner" <daniel.faulkner@scansoft.com>
Subject: RE: No con call
Date: Mon, 27 Nov 2000 15:44:58 -0500

This is a partially processed e-mail
```

If a message is passed into the preprocessor in MIME format, then each MIME boundary is located, and its content type is identified. If there are attachments to the message, the listener is notified what the media type and file name are (e.g., “There is an audio file called hello.wav attached to the message”). File attachments can only be identified if the message is passed in MIME format. Below is an example of a MIME format message with a text file attachment.

```
From andrew.lowry@scansoft.com Fri Nov 24 04:04:16 2000
Received: from [63.113.17.11] by scansoft.com (3.2) with ESMTP id
MBBE7A428003A4004315F3F71110BB50E0; Fri Nov 24 04:03:52
2000
Received: from vishnu.scansoft.com (mailhost.scansoft.com
[206.234.64.17]) by scansoft.com (8.9.3+Sun/8.9.3) with ESMTP id
HAA21041 for < dan.faulkner@scansoft.com>; Fri, 24 Nov 2000
07:05:59 -0500 (EST) Received: from scansoft.com ([10.6.70.30])
by vishnu.scansoft.com (8.9.3+Sun/8.9.3) with ESMTP id
HAA00745
for <dan.faulkner@scansoft.com>; Fri, 24 Nov 2000 07:03:50 -0500
(EST) Message-ID: <3A1E5925.11C0CC@scansoft.com>
Date: Fri, 24 Nov 2000 07:03:49 -0500
From: Andrew Lowry <andrew.lowry@scansoft.com>
X-Mailer: Mozilla 4.7 [en] (WinNT; I)
X-Accept-Language: en
MIME-Version: 1.0
To: dan.faulkner@scansoft.com
```

Chapter XI

```
Subject: here it is!
Content-Type: multipart/mixed;
boundary="D9A5530EC 68939F7E9BE4970"

This is a multi-part message in MIME format.
D9A5530EC 68939F7E9BE4970 Content-Type: text/plain;
charset=us-ascii Content-Transfer-Encoding: 7bit

Here is the file you asked for. Drew
D9A5530EC 68939F7E9BE4970 Content-Type: text/plain;
charset=us-ascii; name="dan.txt"
Content-Transfer-Encoding: 7bit
Content-Disposition: inline;
filename="dan. txt"
blah blah blah blah blah...
D9A5530EC 68939F7E9BE4970
```

Default behavior

For either message format, the Speechify E-mail Pre-processor identifies the message header and message body, and tries to identify address/signature blocks (although this can be difficult if users employ arbitrary formatting for their address/ signature blocks).

Header processing

Discarding header lines

All lines in the header are discarded except the from, date, and subject lines. Each header line is terminated with a period. The expansions of the From, Date and Subject lines are specified in the substitution dictionary. The date in the date line is expanded. The year and time are discarded. For example:

Input	**Preprocessed text**
Date: Tue, 24 Oct 2000 07:03:49 –0500	Your message arrived on Tuesday, twenty fourth October

Chapter XI

Reading From lines

If the From line contains an e-mail address and a real name, the real name is read:

Input	Preprocessed text
From: Dan Faulkner dan.faulkner@scansoft.com	Your message is from Dan Faulkner

Subject line abbreviations

In the subject line, abbreviations like FW and RE can be expanded using the substitution dictionary, e.g.:

Input	Preprocessed text
Subject: RE: Questions from customers	The subject line says the message is a reply about questions from customers

Here "Subject:" was replaced with *The subject line says,* and "RE:" was replaced with *the message is a reply about.*

When expanding the RE: and FW: strings, note that they frequently occur multiple times in a single subject line. For this reason, you should either expand them to something that makes sense when read more than once, or add multiple occurrences to the substitution dictionary, and define a single expansion for them, e.g.:

- RE:, the message is a reply
- FW: the message was forwarded
- FW:FW: the message was forwarded twice
- RE:FW:FW: this message is a reply to a message that was forwarded twice.

Body processing

Discarding data

In the message body, UUencoded data and octal data are skipped. Within the body, an empty line is considered to be a paragraph break, and if the last line of the paragraph doesn't end with a period, then a period is inserted, e.g.:

Original text	Pre-processed text
Hi How are you?	Hi. How are you?

Non-alphanumeric, non-space strings of more than three characters are deleted if they haven't been matched and substituted by the e-mail substitution dictionary. In this example, the lines of dashes have been deleted, so that the listener does not hear "dash dash dash dash dash…":

Original text	**Pre-processed text**
--------------------------------------- All or some of this message may be privileged information. If you are not the intended recipient, please discard this message and any attachments now	All or some of this message may be privileged information. If you are not the intended recipient, please discard this message and any attachments now

Multiple punctuation marks

Multiple punctuation marks resolve to a single punctuation mark, e.g.:

Original text	**Pre-processed text**
What!???????????!!!!	What!

Embedded e-mail messages

The listener is notified of embedded e-mail messages (only the from line is read from embedded messages). The listener is also notified at the beginning and end of portions of text that have been indented with greater than (>), e.g.:

Original text	**Pre-processed text**
>What are you doing on Thursday? I don't know yet.	This next section of text is indented. What are you doing this Thursday? That's the end of the section that was indented. I don't know yet.

These notifications are specified, and can therefore be customized, in the e-mail substitution dictionary. (See chapter "Using the E-mail Substitution Dictionary" for details.)

Chapter XI

Signature processing

The Internet standard for indicating that a signature/address block is next in the text is the sequence --\n. If this sequence is found in the message body, the preprocessor assumes that the following text is an address block, until the next empty line. For example:

--

tel. +44 0803 123456 http://www.scansoft.com

In addition to this assumption, there is a heuristic process employed that looks for the following strings at the beginning of a line:

- ph
- tel
- fax
- pgr
- www
- mail
- http
- work
- home
- email
- e-mail

If any of these are found as the first alphabetic strings on two consecutive lines, then until the next empty line, the text is treated as an address block, e.g.:

Original text	Preprocessed text
+++++++++++++++++++++++ ++++ Tel. +44 0803 123456 + Fax +44 0803 654321 + www.scansoft.com + +++++++++++++++++++++++ +++++++	Telephone, +44 0803 123456. Fax, +44 0803 654321. www.scansoft.com

MIME format

File attachments and media types are only identifiable in MIME format messages. The user can define what should be spoken when a file attachment is found (e.g., “There is an attachment to this message”),

and the name of the file is read (e.g., "expenses.xls"). Otherwise, MIME messages are treated the same as partially processed messages.

Modes

You can control the behavior of the e-mail preprocessor through the use of modes. The user can set the following modes using the SWIemailProcess( ) function. (See chapter "API Reference" for more info about this function)

Mode	**Function**
DATE	Read the Date line from the header
FROM	Read the From line from the header
SUBJECT	Read the Subject line from the header
BODY	Read the message body
ADDRESS	Read any address/signature blocks
MIME_FORMAT	The message is in MIME format

At least one of DATE, FROM, SUBJECT, BODY, ADDRESS must be selected. If the message is in MIME format, then MIME_FORMAT should be combined with the other selections. Modes are combined by using bitwise OR. Here are some examples:

Mode	**Function**
DATE|BODY	Read the date line and the message body
BODY|ADDRESS	Read the body and any address/signature blocks
FROM|SUBJECT	Read the from line and the subject line

Thus, for example, if you don't want to hear the address/signature blocks from a message, don't select ADDRESS.

MIME_FORMAT deserves special mention because it specifies the input rather than the desired output. If you enter a multipart MIME format message and you don't specify MIME_FORMAT, the speech output contains the MIME section boundaries, making it difficult to follow.

If you enter a message that is not in MIME format, and you do specify MIME_ FORMAT, the message may be delivered in an unexpected way. (For example: the body may be skipped altogether and encoded data from file attachments is read out character by character.)

If there are no attachments, the message is read identically whether MIME_ FORMAT is on or not.

Chapter XI

Using the E-mail Substitution Dictionary

This paragraph explains the layout and use of the e-mail substitution dictionary.

In This Paragraph

- Dictionary entries
- Comments and escapes
- Notifications

File format

The e-mail substitution dictionary file is an ASCII text file with one entry per line. A default named "Email.dic" is supplied in the SDK's bin directory. You may supplement that or replace it entirely. (See chapter "API Reference" for more info)

The dictionary is split into five sections:

- Header Entries for substitutions that should only take place in the header
- Body Entries for substitutions that should only take place in the message body
- Address Entries for substitutions that should only take place in the address block
- Mime Entries for substitutions that should only take place in a Mime section boundary
- Global Entries for substitutions that should take place everywhere

The user can decide to expand the same string differently depending on whether it is found in the header or the body, or only expand a string if it is found in a specific section of the message, and ignore it if it appears anywhere else.

The entries in the substitution dictionary do not need to be sorted.

Chapter XI

Dictionary entries

Dictionary entries are of the form

TARGET,SUBSTITUTION

The target is the string to be replaced, and the substitution is the string to replace the target. The target and the substitution are separated by a comma, e.g.:

RE:,Reply

This entry indicates that the e-mail pre-processor replaces the string *RE:* with the string *Reply*. The entries have the same syntax for every section. If an invalid entry is passed in, it is ignored and an error message is logged to stderr. An invalid entry is a line that does not contain a comma and at least one character on each side of the comma (i.e., a target and a substitution).

A target can be any string of any sequence of characters. Multiple words and tokens separated by white spaces are permitted. For example:

George Bush, the former president George W. Bush, the new president

The e-mail pre-processor replaces the longest string possible from left to right. For example, given the hypothetical dictionary entries:

John, I
John Smith, my grandfather
John Smith Jr., my father

Input string	Dictionary output
John Smith Jr. entered the room.	my father entered the room.

Although both *John* and *John Smith* match the input, *John Smith Jr.* is the longest match.

NOTE

When a target has been identified, it is replaced by everything in the substitution. Be careful not to add a space at the end of the substitution by mistake, as it may have consequences for subsequent processing.

For example, suppose we had the following dictionary entry, and accidentally placed a space after the expansion:

SSFT,scansoft

In this case, a web address *www.ssft.com* would be expanded to: *www.scansoft .com*.

This spurious space character would prevent the web address from being processed correctly at a later stage – it would be read as *www dot ScanSoft (pause)com* instead of *www dot ScanSoft dot com*.

By default, the substitution dictionary entries are matched case-insensitively. If you want an entry to be matched case-sensitively, you should append an asterisk to the target. For example, given these dictionary entries:

Tue,Tuesday Wed*,Wednesday

Input string	Dictionary output
The couple wed on tue.	The couple wed on Tuesday.
The couple wed on Wed.	The couple wed on Wednesday.

If you want to use an asterisk as the last character of a genuine substitution, you must escape it, by preceding it with a backslash, for example, to match the input *gold** (case insensitive), use this dictionary entry:

gold*,gold star

Comments and escapes

The e-mail substitution dictionary supports comments. Any line that begins with# is taken as a comment. If you want to use a hash or a comma as part of the TARGET string, escaped it with a backslash. Therefore, the backslash must be escaped too:

Original text	Pre-processor behavior
####This is a comment	Not loaded into memory
\#3,Number Three	Replace *#3* with *Number Three*
Mon\,,Monday	Replace *Mon,* with *Monday*
C:\\Temp, my temp drive	Replace C:\Temp with *my temp drive*

Notifications

Certain events can only be spotted inside the code (i.e., a simple dictionary look-up sometimes is not enough). Examples are:

- UUencoded data
- octal data
- start of indented text
- end of indented text
- end of the message

When the e-mail pre-processor spots these, it inserts an upper case constant into the text before substitutions are performed. The constants that are inserted are self explanatory. They are:

Constant	Meaning
!UUENCODED_DATA_NOTIFY!	UU encoded data found
!OCTAL_DATA_NOTIFY!	Octal data found
!INDENT_NOTIFY!	Start of indented text found
!END_OF_INDENT_NOTIFY!	End of indented text found
!END_MSG_NOTIFY!	End of message found

The following constants are also inserted when file attachments are found in MIME format messages:

- !IMAGE_FILE_NOTIFY!
- !AUDIO_FILE_NOTIFY!
- !APPLICATION_FILE_NOTIFY!
- !VIDEO_FILE_NOTIFY!
- !TEXT_FILE_NOTIFY!

Specify substitution text for these in the substitution dictionary. (The supplied Email.dic file contains appropriate defaults.) This means you can define what you want RealSpeak to say when any of the above events occur. If you want it to say nothing, don't put anything after the delimiter in the dictionary; e.g., if you don't want the listener to be notified that, say, the end of a message had been reached, change the substitution dictionary entry from this:

!END_MSG_NOTIFY!*,That's the end of the message.

To this:

!END_MSG_NOTIFY!*,

NOTE

If you remove these constants from the dictionary, the constant names are deleted by the e-mail pre-processor. This prevents the internal notifications from being read out loud.

Chapter XI

Here is an example of a legal dictionary.

```
\!HEADER ENTRIES
Date:,The message arrived on
From:,The message is from
FW:,the message was forwarded to you, and it's about
RE:, the message is a reply, and it's about

\!MIMEPART ENTRIES

######## RealSpeak internal ########
!IMAGE_FILE_NOTIFY!,There's an image file attached.
!AUDIO_FILE_NOTIFY!,There's an audio file attached.
!APPLICATION_FILE_NOTIFY!,There's an application attached.
!VIDEO_FILE_NOTIFY!,There's a video file attached.
!TEXT_FILE_NOTIFY!,There's a text file attached.

######## Generic ########
.txt, dot text.
.doc, dot doc.
\!BODY ENTRIES
\!GLOBAL ENTRIES

######## RealSpeak internal ########
!UUENCODED_DATA_NOTIFY!,I'll have to skip the next section.
!OCTAL_DATA_NOTIFY!,I'll have to skip the next section.
!INDENT_NOTIFY!,This next section of text is indented.
!END_OF_INDENT_NOTIFY!,That's the end of the indented section.
!END_MSG_NOTIFY!,That's the end of the message.

######## Generic ########
ASAP,as soon as possible
BTW,by the way
FYI,for your information,
WRT,with regard to
```

Chapter XI

API Reference

All API function prototypes, types, error codes, and constants are located in the header file SWIEmail.h.

In This Paragraph

- Calling convention
- Result codes
- SWIemailInit( )
- SWIemailProcess( )
- SWIemailTerm( )

Calling convention

The calling convention is dependent on the operating system, and is defined in the SWIEmail.h header file.

Under NT:

```
#define SWIAPI __stdcall
```

Under Unix:

```
#define SWIAPI
```

Chapter XI

Result codes

The following result codes are defined in the enum SWIemailResult in SWIEmail.h.

Code	Description
SWIemail_SUCCESS	The function completed successfully
SWIemail_BAD_FILE_FORMAT	The substitution dictionary doesn't contain all 5 section headings
SWIemail_EMPTY_MESSAGE	The input string is empty
SWIemail_ERROR	There was an error in the API.
SWIemail_FATAL_EXCEPTION	(NT only.) A crash occurred within the SWIemail library. You should shut down the application.
SWIemail_FILE_NOT_FOUND	Unable to locate the substitution dictionary
SWIemail_MEM_REQUEST	The output string is bigger than the input buffer
SWIemail_UNINITIALIZED	SWIEmailProcess( ) was called before SWIEmailInit( )

Chapter XI

SWIemailInit( )

Mode: Synchronous

Initialize the substitution dictionary.

```
SWIemailResult SWIAPI SWIemailInit(const char *FilePath)
```

Parameter	Description
FilePath	Path to the substitution dictionary

Notes

This must be the first API function called for the e-mail pre-processor.

If you pass in your own FilePath, you can call the dictionary whatever you want. If FilePath is NULL, then SWIemailInit( ) checks for the existence of the environment variable SWITTSSDK. If it exists, SWIemailInit( ) tries to load Email.dic from the path SWITTSSDK/bin. If the variable doesn't exist, SWIemailInit( ) tries to load Email.dic from the current working directory.

Return code	Meaning for SWIemailInit( )
SWIemail_SUCCESS	SWIemailInit()returned successfully
SWIemail_BAD_FILE_FORMAT	The file does not contain the section headings for the HEADER, BODY, ADDRESS, MIMEPART, and GLOBAL sections.
SWIemail_ERROR	Error opening the file
SWIemail_FILE_NOT_FOUND	SWIemailInit() did not find a file at all

Chapter XI

SWIemailProcess( )

Mode: Synchronous

Processes an e-mail message.

SWIemailResult SWIAPI SWIemailProcess(const char *Message, int *BufferSize, unsigned char Modes)

Parameter	Description
Message	The e-mail message as a null-terminated string. The output is copied into this buffer.
BufferSize	Size of the input buffer. Its contents are changed if the output string is bigger then the input buffer.
Modes	Modes through combinations of the mode variables defined in SWIEmail.h.(See “Modes” in chapter “)

Return code	Meaning for SWIemailProcess( )
SWIemail_SUCCESS	SWIemailProcess() returned successfully
SWIemail_EMPTY_MESSAGE	Input string is empty.
SWIemail_ERROR	Memory error in the API.
SWIemail_MEM_REQUEST	The output string is longer than the input buffer, and the value of the contents of BufferSize is changed to the required size.
SWIemail_UNINITIALIZED	SWIemailProcess()called before SWIemailInit( ).

Notes

The parameter BufferSize tells the function how long the input buffer is. If the function returns SWIemail_MEM_REQUEST, the application should reallocate the input buffer to the appropriate size and call SWIemailProcess( ) again.

Chapter XI

SWIemailTerm( )

Mode: Synchronous

Frees resources used by the preprocessor library.

```
SWIemailResult SWIAPI SWIemailTerm( )
```

SWIemailTerm( ) must be the last function called. If it is called before calling SWIemailInit( ), it returns SWIemail_UNINITIALIZED. It returns SWIemail_ SUCCESS on completion.

RealSpeak Telecom
Software Development Kit

Appendices

Programmer's Guide

Appendix A

Appendices

Appendix: TTSPARM member values

The following table shows the list of acceptable values for each member in the TTSPARM structure specified when a RealSpeak engine instance is initialized via the TtsInitialize(Ex) function:

TTSPARM member	Acceptable values
nLanguage	TTS_LANG_US_ENGLISH TTS_LANG_SPANISH TTS_LANG_FRENCH TTS_LANG_NETHERLANDS_DUTCH TTS_LANG_DUTCH TTS_LANG_BRITISH_ENGLISH TTS_LANG_GERMAN TTS_LANG_ITALIAN TTS_LANG_JAPANESE TTS_LANG_KOREAN TTS_LANG_EGYPTIAN_ARABIC TTS_LANG_MANDARIN_B5 TTS_LANG_BRAZILIAN_PORTUGUESE TTS_LANG_RUSSIAN TTS_LANG_MEXICAN_SPANISH TTS_LANG_BELGIAN_DUTCH TTS_LANG_SWEDISH TTS_LANG_NORWEGIAN TTS_LANG_MANDARIN_GB TTS_LANG_AUSTRALIAN_ENGLISH TTS_LANG_CANADIAN_FRENCH TTS_LANG_CANTONESE_B5 TTS_LANG_CANTONESE_GB TTS_LANG_DANISH TTS_LANG_PORTUGAL_PORTUGUESE TTS_LANG_POLAND_POLISH TTS_LANG_ARMENIA_ARMENIAN TTS_LANG_UKRAINIAN TTS_LANG_GREEK TTS_LANG_VIETNAMESE TTS_LANG_MALAY

Appendix A

TTSPARM member	Acceptable values
	TTS_LANG_PAKISTAN_URDU TTS_LANG_INDONESIA_BAHASA TTS_LANG_IRAN_FARSI TTS_LANG_BELARUSIAN TTS_LANG_CZECH TTS_LANG_HUNGARIAN TTS_LANG_INDIA_TAMIL TTS_LANG_THAILAND_THAI TTS_LANG_TURKISH TTS_LANG_TAIWANESE TTS_LANG_INDIA_HINDI TTS_LANG_TAIWAN_MANDARIN_B5 TTS_LANG_TAIWAN_MANDARIN_GB TTS_LANG_BELGIAN_FRENCH TTS_LANG_INDIAN_ENGLISH
nOutputType	TTS_LINEAR_16BIT TTS_MULAW_8BIT TTS_ALAW_8BIT
nFrequency	TTS_FREQ_8KHZ TTS_FREQ_11KHZ TTS_FREQ_22KHZ
nVoice	TTS_RS_VOICE_FEMALE TTS_RS_VOICE_MALE TTS_3000_VOICE_FEMALE TTS_3000_VOICE_MALE TTS_RS_VOICE_FEMALE2 TTS_RS_VOICE_MALE2 TTS_RS_VOICE_FEMALE3 TTS_RS_VOICE_MALE3 TTS_VOICE_USE_STRING

Appendix B

Appendix: RealSpeak API Function Directory

The following table shows an alphabetical list of the RealSpeak v4 API functions. The RealSpeak v3.5 functions for which exists a new variant are also supported, but their name is put in-between brackets.

Use this Function…	To Perform this Task…
TtsCreateEngine	Create a TTS engine instance on the server[1].
TtsDisableUsrDictEx (TtsDisableUsrDict)	Disable a dictionary instance for a channel.
TtsDisableUsrDictsEx	Disable all dictionary instances for a channel
TtsEnableUsrDictEx (TtsEnableUsrDict)	Enable a dictionary instance for a channel.
TtsGetG2PDictList	Get the list of custom G2P dictionaries on the system for the current language.
TtsGetG2PDictTotal	Retrieve the total number of custom G2P dictionaries on the system for the current language.
TtsGetParams	Get the values of a parameter. list
TtsGetParam	Get the value of a parameter.
TtsInitializeEx (TtsInitialize)	Create an instance of the TTS engine.
TtsLoadG2PDictList	Load a list of G2P dictionaries
TtsLoadUsrDictEx (TtsLoadUsrDict)	Load a user dictionary to be used by the engine.
TtsProcessEx (TtsProcess)	Convert input data (text) into output data (speech).
TtsRemoveEngine	Remove an instance of an engine from the server.
TtsSetParams	Set the values for multiple parameters.
TtsSetParam	Set the values for the document type, rate, and volume.
TtsStop	Stops the Text-To-Speech process.
TtsUninitialize	Free all resources allocated to an engine.
TtsUnloadG2PDictList	Unload a list of G2P dictionaries
TtsUnloadUsrDictEx	Unload a dictionary from memory; frees

[1] This is no longer the case with v4; for the moment it is a dummy function, reserved for future use.

Appendix B

Use this Function…	To Perform this Task…
(TtsUnloadUsrDict)	memory resources.

Appendix C

Appendix: Running a TTS server as a service (Windows only)

A service in Microsoft® Windows is a program that runs whenever the computer is running the operating system. It does not require a user to be logged on. For more information on how to run and configure Windows services, refer to the Microsoft documentation that comes with your operating system.

Included with the Telecom RealSpeak/Host SDK is a ttsserver_service.exe, which is a version of ttsserver.exe that can run as a Windows service.

If ttsserver_service is not already registered as a service, then you can do so with the following command:

ttsserver_service –i

To unregister, use the –u parameter:

ttsserver_service –u

To work properly, the TTS server must be configured using $SSFTTTSSDK/config/ttsserver.xml. For information on that configuration file, see the “Configuration Files” section in the “User Configuration” chapter..

Appendix D

Appendix: Port density simulator

The port density simulator can help a customer identify the hardware needed in order to run a certain amount of RealSpeak channels in real-time. Real-time means that any generated PCM, on any of the active channels, can be spoken as soon as possible without any interruptions.

The simulator starts out using the input parameters given on the command line. In its simplest form a thread will be created and will process the requested language, output type, voice, output buffer size and input text.

When processing, the E-mail pre-processor is not enabled and no user dictionaries are loaded. Immediately when the thread finishes a result will be returned reporting if the current thread was executed in real-time or not. If it took less time to generate the PCM then it would take time to speak the text then it's considered real-time.

The simulator will continue to increase the amount of active threads, one by one, until the system reaches a point were the overall system is not real-time anymore. This point is reached when the average thread isn't real-time. A real-time time index (RTI) is attached to each executed thread and a number below 100 means not real-time and a number equal or greater then 100 as real-time. At different steps of the program data will be collected in a formatted output file (name is user-defined).

The generated output file is a comma-separated file that can be imported directly into a spreadsheet.

Syntax:

Usage: density.exe LangId VoiceId OutType StartAt# OutSize LibDir InName OutName

Where:

LangId: The language define from lh_ttsso.h
VoiceId: The voice define from lh_ttso.h
OutType: The output type define from lh_ttso.h
StarAt#: Sets the minimum amount of active threads (min 1)
OutSize: Sets the output buffer size (must be even)
LibDir: Points to the engine directory
InName: Name of the input text file
OutName: Name of the output file (.csv file)

Example: density.exe 0 0 0 1 2048 ./speech ./input.txt ./output.csv

Appendix E

Appendix: Copyright and Licensing for third party software

The Telecom RealSpeak/Host SDK utilizes certain open source software packages. Copyright and licensing information for these packages are included in this section.

ADAPTIVE Communication Environment (ACE)

Copyright and Licensing Information for ACE(TM) and TAO(TM)

[1]ACE(TM) and [2]TAO(TM) are copyrighted by [3]Douglas C. Schmidt and his [4]research group at [5]Washington University, Copyright (c) 1993-2001, all rights reserved. Since ACE and TAO are [6]open source, [7]free software, you are free to use, modify, and distribute the ACE and TAO source code and object code produced from the source, as long as you include this copyright statement along with code built using ACE and TAO.

In particular, you can use ACE and TAO in proprietary software and are under no obligation to redistribute any of your source code that is built using ACE and TAO. Note, however, that you may not do anything to the ACE and TAO code, such as copyrighting it yourself or claiming authorship of the ACE and TAO code, that will prevent ACE and TAO from being distributed freely using an open source development model.

ACE and TAO are provided as is with no warranties of any kind, including the warranties of design, merchantability and fitness for a particular purpose, noninfringement, or arising from a course of dealing, usage or trade practice. Moreover, ACE and TAO are provided with no support and without any obligation on the part of Washington University, its employees, or students to assist in its use, correction, modification, or enhancement. However, commercial support for ACE and TAO are available from [8]Riverace and [9]OCI, respectively. Moreover, both ACE and TAO are Y2K-compliant, as long as the underlying OS platform is Y2K-compliant.

Washington University, its employees, and students shall have no liability with respect to the infringement of copyrights, trade secrets or any patents by ACE and TAO or any part thereof. Moreover, in no event will Washington University, its employees, or students be liable for any lost revenue or profits or other special, indirect and consequential damages.

The [10]ACE and [11]TAO web sites are maintained by the [12]Center for Distributed Object Computing of Washington University for the development of open source software as part of the [13]open source software community. By submitting comments,

suggestions, code, code snippets, techniques (including that of usage), and algorithms, submitters acknowledge that they have the right to do so, that any such submissions are given freely and unreservedly, and that they waive any claims to copyright or ownership. In addition, submitters acknowledge that any such submission might become part of the copyright maintained on the overall body of code, which comprises the [14]ACE and [15]TAO software. By making a submission, submitter agrees to these terms. Furthermore, submitters acknowledge that the incorporation or modification of such submissions is entirely at the discretion of the moderators of the open source ACE and TAO projects or their designees.

The names ACE (TM), TAO(TM), and Washington University may not be used to endorse or promote products or services derived from this source without express written permission from Washington University. Further, products or services derived from this source may not be called ACE(TM) or TAO(TM), nor may the name Washington University appear in their names, without express written permission from Washington University.

If you have any suggestions, additions, comments, or questions, please let [16]me know.

[17]Douglas C. Schmidt

Back to the [18]ACE home page.

References

1. http://www.cs.wustl.edu/~schmidt/ACE.html
2. http://www.cs.wustl.edu/~schmidt/TAO.html
3. http://www.cs.wustl.edu/~schmidt/
4. http://www.cs.wustl.edu/~schmidt/ACE-members.html
5. http://www.wustl.edu/
6. http://www.opensource.org/
7. http://www.gnu.org/
8. http://www.riverace.com/
9. file://localhost/home/cs/faculty/schmidt/.www-docs/www.ociweb.com
10. http://www.cs.wustl.edu/~schmidt/ACE.html
11. http://www.cs.wustl.edu/~schmidt/TAO.html
12. http://www.cs.wustl.edu/~schmidt/doc-center.html
13. http://www.opensource.org/
14. http://www.cs.wustl.edu/~schmidt/ACE-obtain.html
15. http://www.cs.wustl.edu/~schmidt/TAO-obtain.html
16. mailto:schmidt@cs.wustl.edu
17. http://www.cs.wustl.edu/~schmidt/
18. file://localhost/home/cs/faculty/schmidt/.www-docs/ACE.html

Appendix E

Apache Group

The Apache Software License, Version 1.1 Copyright (c) 1999 The Apache Software Foundation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end user documentation included with the redistribution, f any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.

4. The names "Xerces" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
==

Appendix E

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation and was originally based on software copyright (c) 1999, International Business Machines, Inc., http://www.ibm.com. For more information on the Apache Software Foundation, please see <http://www.apache.org/>.

The Flite Speech Synthesis System

Language Technologies Institute
Carnegie Mellon University
Copyright (c) 1999-2003
All Rights Reserved.
http://cmuflite.org

Dinkumware C++ Library for Visual C++

Developed by P.J. Plauger
Copyright (c) 1992-2000 by P.J. Plauger
Dinkumware, Ltd.
398 Main Street
Concord MA 01742

RSA Data Security, Inc. MD5 Message-Digest Algorithm

Copyright (c) 1991-1992, RSA Data Security, Inc. Created 1991. All Rights
Reserved.

ICU

http://oss.software.ibm.com/icu/
COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1995-2003 International Business Machines Corporation and others
All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

Appendix E

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT
OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL
INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING
FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

All trademarks and registered trademarks mentioned herein are the property of their respective owners.

PCRE

PCRE LICENCE

PCRE is a library of functions to support regular expressions whose syntax and semantics are as close as possible to those of the Perl 5 language.
Release 5 of PCRE is distributed under the terms of the "BSD" licence, as specified below. The documentation for PCRE, supplied in the "doc" directory, is distributed under the same terms as the software itself.

Written by: Philip Hazel <ph10@cam.ac.uk>

University of Cambridge Computing Service,
Cambridge, England. Phone: +44 1223 334714.

Copyright (c) 1997-2004 University of Cambridge
All rights reserved.

Redistribution and use in source and binary forms, with or without

Appendix E

modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the University of Cambridge nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.

Appendix F

Appendix: RealSpeak Languages

The following table shows a list of all the RealSpeak languages, sorted according to the ISO language code. It also specifies the 3-letter RealSpeak language code (which is used to specify the language in the header of user dictionaries and rulesets) and the native character set. Note that for each given language, TTS input encoded in another supported (coded) character set is converted to the native character set for that language before it is processed internally. See the RealSpeak User's Guide for each language for more details.

Appendix F

Language name	ISO language code	RealSpeak language code	Native character set
Danish	da-DK	DAD	windows-1252
Swiss German	de-CH	GEC	windows-1252
German	de-DE	GED	windows-1252
Australian English	en-AU	ENA	windows-1252
British English	en-GB	ENG	windows-1252
Indian English	en-IN	ENI	windows-1252
American English	en-US	ENU	windows-1252
Spanish	es-ES	SPE	windows-1252
Mexican Spanish	es-MX	SPM	windows-1252
Basque	eu-ES	BAE	windows-1252
Belgian French	fr-BE	FRB	windows-1252
Canadian French	fr-CA	FRC	windows-1252
Swiss French	fr-CH	FNC	windows-1252
French	fr-FR	FRF	windows-1252
Swiss Italian	it-CHC	ITC	windows-1252
Italian	it-IT	ITI	windows-1252
Japanese	ja-JP	JPJ	Shift-JIS
Korean	ko-KR	KOK	windows-949
Korean	kr-KR	KOK	windows-949
Belgian Dutch	nl-BE	DUB	windows-1252
Dutch	nl-NL	DUN	windows-1252
Norwegian	no-NO	NON	windows-1252
Polish	pl-PL	PLP	windows-1250
Brazilian Portuguese	pt-BR	PTB	windows-1252
Portuguese	pt-PT	PTP	windows-1252
Russian	ru-RU	RUR	windows-1251
Swedish	sv-SE	SWS	windows-1252
Mandarin Chinese[1]	zh-CN	MNC	windows-936 (GB2312) or windows-950 (Big5)
Hong Kong Cantonese[2]	zh-HK	CAH	windows-950 (Big5)

[1] "Mandarin Chinese GB" and "Mandarin Chinese B5" are also supported as valid languages; they also specify the native character set to be used.
[2] "Hong Kong Cantonese B5" is also supported as a valid language; it also specifies the native character set to be used.

Appendix G

Appendix: Tips for using RealSpeak

Operating System Restrictions

Each instance of RealSpeak requires a number of file handles which are used for accessing among others the language database. Some operating systems, such as Microsoft Windows, have a default limit of file handles per process. If you have a large number of RealSpeak instances, the application can run out of file handles. In order to avoid this problem, ScanSoft recommends you to set the amount of file handles to an appropriate value. For Microsoft Windows, you can set this value by having your application issue the _setmaxstdio() C-runtime call. See your compiler or operating system manual for more information.
For Unix, the number of file handles can be increased by means of the limit/ulimit commands. For more information about these commands, refer to the man pages or to the compiler manual.

Optimal Audio Buffer size

For optimal performance in client-server mode, the audio buffer size should be set to 4k (4096) bytes. This buffer is provided by the application and is specified via the return value of the TTSDESTCB typed Destination call-back function.

Limiting delays when internet fetching is used

When content such as input texts, user dictionaries and rulesets are located on a Web server, this can result in delays when the content is fetched for the first time. But since the internet fetch library uses a (configurable) cache, the download time will be minimal if the cache has been configured well (big enough, reasonable cache entry expiration time) and the cache has been warmed up.
To warm up the cache, the application could perform a number of dummy speak requests. For input texts, the content will already be cached before the Destination call-back is called for the first time. So during the warmup, the application can call the TtsStop function from that moment on to speed up the warmup.
Audio content specified via the SSML <audio> tag, is always fetched on message (normally a sentence) boundaries, but not necessarily before the first call to the Destination call-back. User dictionaries and rulesets can be loaded and unloaded to obtain a copy in the cache without consuming RAM memory. If RAM usage is not a problem, load them as soon as possible.

Appendix G

Binary versus textual user dictionaries

If the application repeatedly loads/unloads one or more user dictionaries (such as load it for a single speak request here and there), then a binary dictionary loads faster. But once loaded, the run-time access speed is the same. When loading all dictionaries at startup, it doesn't really matter. When default dictionaries are specified in the server configuration file (via the <default_dictionaries> element), they are loaded/unloaded when the language (or voice) parameter is updated via TtsSetParam(s) or when an engine instance is created. The User Dictionary Editor (UDE) is the only way to obtain a binary dictionary. Please check out the online help documentation that comes with the UDE for detailed instructions.